L'Ultima Cena

Leonardo era molto attivo quando lavorava sull'Ultima Cena, il suo più famoso affresco: uno dei suoi primi compiti quotidiani era quello di decidere quali colori a tempera usare nel resto del giorno di lavoro. Egli necessitava di molti colori ma poteva tenerne solo un numero limitato sul suo ponteggio. Il suo assistente aveva l'incarico, tra le altre cose, di arrampicarsi sul ponteggio per portargli i colori e scendere per rimetterli a posto su un apposito scaffale a terra.

In questo task, dovrai scrivere due programmi separati per aiutare l'assistente. Il primo programma riceverà le istruzioni di Leonardo (una sequenza di colori di cui Leonardo avrà bisogno durante il giorno), e creerà una *breve* stringa di bit, chiamata suggerimento. Mentre l'assistente gestisce le richieste di Leonardo durante il giorno, egli non ha accesso alle richieste future di Leonardo, ma solo al suggerimento prodotto dal tuo primo programma. Il secondo programma riceverà il suggerimento, e successivamente riceverà e eseguirà le richieste di Leonardo online (cioè una alla volta). Il programma deve essere in grado di capire cosa il suggerimento significa e usarlo per fare scelte ottimali. Segue sotto una spiegazione con maggiori dettagli.

Spostamento dei colori tra lo scaffale e il ponteggio

Considereremo uno scenario semplificato. Supponi che ci siano N colori numerati da 0 a N - 1, e che ogni giorno Leonardo chieda al suo assistente un nuovo colore esattamente N volte. Sia C la sequenza delle N richieste di colori fatte da Leonardo. Possiamo immaginare C come una sequenza di N numeri, ognuno compreso tra 0 e N - 1 (estremi inclusi). Nota che alcuni colori potrebbero non essere presenti in C, mentre altri potrebbero apparire più volte.

Il ponteggio è sempre pieno e contiene K degli N colori, con $K < N$. Inizialmente, il ponteggio contiene i colori da 0 a K - 1 (estremi inclusi).

L'assistente gestisce le richieste di Leonardo una alla volta. Ogni volta che il colore richiesto è *già presente sullo scaffale*, l'assistente può riposare. Altrimenti, egli deve prendere il colore richiesto dallo scaffale e spostarlo sul ponteggio. Ovviamente, non c'è spazio sul ponteggio per il nuovo colore, quindi l'assistente deve scegliere uno dei colori già presenti sul ponteggio e riportarlo dal ponteggio allo scaffale.

La strategia ottimale di Leonardo

L'assistente vuole riposare più volte possibile. Il numero di richieste in cui può riposare dipende dalle sue scelte durante il processo. Più precisamente, ogni volta che l'assistente deve rimuovere un colore dal ponteggio, scelte differenti potrebbero condurre a esiti diversi nel futuro. Leonardo gli spiega come può ottenere il suo obiettivo conoscendo C. La scelta migliore per il colore da rimuovere dal ponteggio si ottiene esaminando i colori attualmente sul ponteggio e le richieste di

colore rimanenti in C. Bisogna scegliere un colore tra quelli attualmente sul ponteggio in base alle seguenti regole:

- Se c'è un colore sul ponteggio che non sarà mai necessario in futuro, l'assistente deve rimuovere quel colore dal ponteggio.
- Altrimenti, il colore rimosso dal ponteggio deve essere quello *che sarà richiesto più avanti in futuro*. (Cioè, per ognuno dei colori sul ponteggio troviamo la sua prima occorrenza futura. Il colore da riporre sullo scaffale è quello che sarà richiesto per ultimo.)

Si può dimostrare che, usando la strategia di Leonardo, l'assistente riposerà più volte possibile.

Esempio 1

Sia N = 4, così che abbiamo 4 colori (numerati da 0 a 3) e 4 richieste. Sia la sequenza di richieste C = (2, 0, 3, 0). Inoltre assumi che K = 2. Cioè, Leonardo può tenere sul ponteggio 2 colori in ciascun istante. Come indicato sopra, inizialmente il ponteggio contiene i colori 0 e 1. Scriveremo il contenuto del ponteggio come segue: [0, 1]. Un possibile modo in cui l'assistente può gestire le richieste è descritto a seguire.

Il primo colore richiesto (numero 2) non è sul ponteggio. L'assistente lo porta e decide di rimuovere il colore 1 dal ponteggio. Il ponteggio corrente è [0, 2].

- Il seguente colore richiesto (numero 0) è già sul ponteggio, quindi l'assistente può riposare.
- Per la terza richiesta (numero 3), l'assistente rimuove il colore 0, trasformando il ponteggio in [3, 2].
- Alla fine, l'ultimo colore richiesto (numero 0) deve essere portato dallo scaffale al ponteggio. L'assistente decide di rimuovere il colore 2, e il ponteggio diventa [3, 0].

Nota che nell'esempio sopra l'assistente non ha seguito la strategia ottimale di Leonardo. La strategia ottimale rimuoverebbe il colore 3 nel terzo passo, così che l'assistente potrebbe riposare nuovamente nel passo finale.

La strategia dell'assistente quando la sua memoria è limitata

La mattina, l'assistente chiede a Leonardo di scrivere C su un pezzo di carta, così che egli possa stabilire e seguire la strategia ottimale. Tuttavia, Leonardo ha la mania di tenere le sue tecniche lavorative segrete, quindi si rifiuta di lasciare il foglio all'assistente. Permette all'assistente solo di leggere C e cercare di ricordarlo.

Sfortunatamente, l'assistente ha una memoria terribile. È capace di ricordare solo al più M bit. In generale, questo gli potrebbe impedire di ricostruire l'intera sequenza C. Quindi, l'assistente deve trovare un modo furbo di calcolare la sequenza di bit che dovrà ricordare. Chiameremo questa sequenza la *sequenza suggeritrice* e la denoteremo con A.

Esempio 2

La mattina, l'assistente può prendere il foglio di Leonardo con la sequenza C, leggere la sequenza, e fare tutte le scelte necessarie. Una cosa che potrebbe considerare è esaminare lo stato del ponteggio

dopo ogni richiesta. Per esempio, usando la strategia (subottimale) descritta nell'Esempio 1, la sequenza di stati del ponteggio sarebbe [0, 2], [0, 2], [3, 2], [3, 0]. (Ricorda che egli sa che lo stato iniziale del ponteggio è [0, 1])

Adesso assumi che abbiamo M = 16, quindi l'assistente è capace di ricordare fino a 16 bit di informazione. Se N = 4, possiamo memorizzare ogni colore usando 2 bit. Di conseguenza, 16 bit sono sufficienti per memorizzare la sequenza di stati del ponteggio descritta sopra. Quindi l'assistente calcola la seguente sequenza suggeritrice: A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`).

Più tardi, l'assistente può decodificare questa sequenza suggeritrice e usarla per fare le sue scelte.

(Ovviamente, con M = 16 l'assistente può anche scegliere di ricordare l'intera sequenza C, usando solo 8 dei 16 bit disponibili. In questo esempio abbiamo solo voluto illustrare che egli può considerare diverse opzioni, senza che rivelassimo una buona soluzione.)

Descrizione del problema

Devi scrivere *due programmi separati* nello stesso linguaggio di programmazione. Questi programmi saranno eseguiti uno dopo l'altro, senza essere in grado di comunicare tra di loro durante l'esecuzione.

Il primo programma sarà quello usato dall'assistente di mattina. Al programma sarà data la sequenza C, ed esso deve calcolare una sequenza suggeritrice A.

Il secondo programma sarà quello usato dall'assistente durante il giorno. Questo programma riceverà la sequenza suggeritrice A, e successivamente deve gestire la sequenza C di richieste di Leonardo. Nota che la sequenza C sarà rivelata a questo programma una richiesta alla volta, e ogni richiesta deve essere eseguita prima di ricevere la successiva.

Più precisamente, nel primo programma devi implementare una singola funzione `ComputeAdvice`(C, N, K, M) che ha come input l'array C di N interi (ognuno compreso in 0, …, N - 1), il numero K di colori nel ponteggio, e il numero M di bit disponibili per il suggerimento. Questo programma deve calcolare una sequenza suggeritrice A che consiste in al più M bit. Il programma deve quindi comunicare la sequenza A al sistema chiamando, per ogni bit di A in ordine, la seguente funzione:

- `WriteAdvice`(B) — aggiungi il bit B in fondo alla corrente sequenza suggeritrice A. (Puoi chiamare questa funzione al più M volte.)

Nel secondo programma devi implementare una singola funzione `Assist`(A, N, K, R). L'input di questa funzione è la sequenza suggeritrice A, gli interi N e K definiti sopra, e la lunghezza effettiva R della sequenza A in bit ($R \leq M$). Questa funzione dovrebbe eseguire la tua strategia proposta per l'assistente, usando le seguenti funzioni che ti sono fornite:

- `GetRequest`() — restituisce il prossimo colore richiesto da Leonardo. (Non viene rivelata alcuna informazione sulle richieste future.)

- `PutBack`(T) — ripone il colore T dal ponteggio sullo scaffale. È richiesto che T sia uno dei colori attualmente sul ponteggio.

Quando eseguita, la tua funzione `Assist` deve chiamare `GetRequest` esattamente N volte, ogni volta ricevendo una delle richieste di Leonardo, in ordine. Dopo ogni chiamata a `GetRequest`, se il colore restituito **non** è sul ponteggio, **devi** anche chiamare `PutBack`(T) con la tua scelta di T. Altrimenti, **è proibito** chiamare `PutBack`. Fare altrimenti è considerato un errore e causerà la terminazione del tuo programma. Ricorda che all'inizio il ponteggio contiene i colori da 0 a K - 1 (estremi inclusi).

Uno specifico test case sarà considerato risolto se le tue due funzioni seguono tutti i vincoli imposti, e se il numero totale di chiamate a `PutBack` è *esattamente uguale* a quello della strategia ottimale di Leonardo. Nota che se esistono più strategie che ottengono lo stesso numero di chiamate a `PutBack`, al tuo programma è permesso di eseguirne una qualunque. (Cioè, non è richiesto di seguire la strategia di Leonardo, se esiste una diversa strategia altrettanto buona.)

Esempio 3

Proseguendo l'Esempio 2, assumi che in `ComputeAdvice` tu abbia calcolato A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Per comunicarlo al sistema, sarebbe necessario effettuare la seguente sequenza di chiamate: `WriteAdvice(0)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(0)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(1)`, `WriteAdvice(1)`, `WriteAdvice(0)`, `WriteAdvice(0)`.

La tua seconda funzione `Assist` sarebbe quindi eseguita, ricevendo la sequenza A di cui sopra, e i valori N = 4, K = 2, and R = 16. La funzione `Assist` deve successivamente eseguire esattamente N = 4 chiamate a `GetRequest`. Inoltre, dopo alcune di queste richieste, `Assist` dovrà chiamare `PutBack`(T) con una opportuna scelta di T.

La seguente tabella mostra una sequenza di chiamate che corrisponde alle scelte (sub-ottimali) dell'Esempio 1. Il trattino indica che non è stata effettuata la chiamata a `PutBack`.

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

Subtask 1 [8 punti]

- $N \leq 5\,000$.
- Puoi usare al più M = 65 000 bits.

Subtask 2 [9 punti]

- $N \leq 100\,000$.
- Puoi usare al più $M = 2\,000\,000$ bits.

Subtask 3 [9 punti]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Puoi usare al più $M = 1\,500\,000$ bits.

Subtask 4 [35 punti]

- $N \leq 5\,000$.
- Puoi usare al più $M = 10\,000$ bits.

Subtask 5 [fino a 39 punti]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Puoi usare al più $M = 1\,800\,000$ bits.

Il punteggio per questo subtask dipende dalla lunghezza R del suggerimento comunicato dal tuo programma. Più precisamente, se R_{max} è la lunghezza massima (tra tutti i casi di test) della sequenza suggeritrice prodotta dalla tua funzione `ComputeAdvice`, il tuo punteggio sarà:

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

Dettagli implementativi

Devi inviare esattamente due files *nello stesso linguaggio di programmazione*.

Il primo file si chiama `advisor.c`, `advisor.cpp` o `advisor.pas`. Questo file deve implementare la funzione `ComputeAdvice` come descritto sopra e può chiamare la funzione `WriteAdvice`. Il secondo file si chiama `assistant.c`, `assistant.cpp` o `assistant.pas`. Questo file deve implementare la funzione `Assist` come descritto sopra e può chiamare le funzioni `GetRequest` e `PutBack`.

Seguono i prototipi di tutte le funzioni.

Programmi 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();
```

Programmi 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;
```

Queste funzioni devono comportarsi come descritto sopra. Ovviamente sei libero di implementare altre funzioni per uso interno. Nel caso di programmi C/C++, le tue funzioni interne devono essere dichiarate `static`, dal momento che verranno linkate dal grader di esempio. Alternativamente, puoi semplicemente evitare di avere due funzioni (una in ogni programma) con lo stesso nome. Le tue sottoposizioni non devono interagire in alcun modo con l'input/output standard, né con nessun altro file.

Mentre sviluppi la tua soluzione, devi anche rispettare le istruzioni seguenti (i templates che trovi nell'ambiente di gara soddisfano già i requisiti indicati).

Programmi C/C++

All'inizio della tua soluzione, devi includere i files `advisor.h` e `assistant.h` rispettivamente nel suggeritore e nell'assistente. Questo si effettua mettendo nel tuo codice la linea:

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

o

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

I due files `advisor.h` e `assistant.h` ti verranno forniti in una directory nel tuo ambiente di gara e verranno anche resi disponibili dall'interfaccia Web. Ti verranno anche forniti (tramite le stesse modalità) codice e scripts per compilare e testare la tua soluzione. In particolare, dopo aver copiato la tua soluzione nella directory con questi scripts, dovrai eseguire `compile_c.sh` o `compile_cpp.sh` (a seconda del linguaggio in cui è il tuo codice).

Programmi Pascal

Devi usare le units `advisorlib` e `assistantlib` rispettivamente nel suggeritore e nell'assistente. Questo si effettua mettendo nel tuo codice la linea:

```
uses advisorlib;
```

o

```
uses assistantlib;
```

I due files `advisorlib.pas` e `assistantlib.pas` ti verranno forniti in una directory nel tuo ambiente di gara e verranno anche resi disponibili dall'interfaccia Web. Ti verranno anche forniti (tramite le stesse modalità) codice e scripts per compilare e testare la tua soluzione. In particolare, dopo aver copiato la tua soluzione nella directory con questi scripts, dovrai eseguire `compile_pas.sh`.

Grader di esempio

Il grader di esempio accetterà input formattato come segue:

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

Per prima cosa il grader eseguirà la funzione `ComputeAdvice`. Questo genererà un file `advice.txt`, che contiene uno ad uno i bits della sequenza suggeritrice, separati da spazi e terminati da un 2.

Quindi eseguirà la tua funzione `Assist` e genererà un output in cui ciascuna linea ha la forma "`R [number]`" oppure "`P [number]`". Le linee del primo tipo indicano una chiamata a `GetRequest()` e la relativa risposta. Le linee del secondo tipo rappresentano una chiamata a `PutBack()` e i colori che si è deciso di riporre. L'output è terminato da una linea del tipo "`E`".

Nota che sul grader ufficiale il tempo di esecuzione del tuo programma potrebbe differire leggermente dal tempo ottenuto sul tuo computer locale. Questa differenza non dovrebbe essere significativa. Ad ogni modo, ti invitiamo ad usare l'interfaccia di test per verificare che la tua soluzione rientri nei limiti di tempo.

Limiti di tempo e memoria

- Limite di tempo: 7 secondi.
- Limite di memoria: 256 MiB.