Das letzte Abendmahl

Als Leonardo am "Letzten Abendmahl", seiner berühmtesten Wandmalerei, arbeitete, war er sehr aktiv: an jedem Tag war eine seiner ersten Arbeiten, zu entscheiden, welche Temperafarben er für den Rest seines Arbeitstages benutzen möchte. Er benötigte viele Farben aber konnte nur eine begrenzte Anzahl davon auf seinem Gerüst aufbewahren. Sein Assistent war unter anderem dafür zuständig, zu ihm aufs Gerüst zu klettern, ihm die benötigten Farben zu übergeben und anschliessend wieder runter zu klettern, um Farben auf dem Boden abzustellen.

In dieser Aufgabe musst du zwei getrennte Programme schreiben, um dem Assistenten zu helfen. Das erste Programm enthält Leonardos Anforderungen (eine Folge der Farben, welche Leonardo während des Tages benötigen wird) und soll einen *kurzen* Plan erstellen, welcher *Advice* genannt wird. Während des Tages wird der Assistent keinen Zugang zu Leonardos zukünftigen Anforderungen haben, sondern nur zum Advice, welcher von deinem ersten Programm erzeugt wurde. Das zweite Programm erhält den Advice und soll dann Leonardos Anforderungen anfragen und bearbeiten und zwar online, d.h. eine nach der anderen. Dieses Programm muss in der Lage sein, zu verstehen, was der Advice bedeutet, und ihn dazu benutzen, optimale Entscheidungen zu treffen. All dies wird nun detailliert beschrieben.

Bewegen von Farben vom Boden auf das Gerüst und zurück

Wir betrachten eine vereinfachte Situation. Nimm an, dass es N verschiedene Farben gibt, nummeriert von 0 bis N - 1, und dass Leonardo seinen Assistenten jeden Tag genau N mal um eine neue Farbe bittet. Es sei C die Folge der N Farbanforderungen. Also können wir uns C als Folge von N Zahlen vorstellen, welche zwischen 0 und N - 1 liegen. Beachte, dass einige Farben in C gar nicht vorkommen können und dass andere mehrfach enthalten sein können.

Das Gerüst ist immer voll gefüllt und enthält K der N Farben, wobei $K < N$ gilt. Zu Beginn sind die Farben 0 bis K - 1 auf dem Gerüst.

Der Assistent bearbeitet Leonardos Anforderungen eine nach der anderen. Immer wenn die angefragte Farbe *bereits auf dem Gerüst* ist, kann der Assistent Pause machen. Andernfalls muss er die angefragte Farbe vom Boden aufheben und aufs Gerüst bringen. Natürlich gibt es auf dem Gerüst keinen Platz für die neue Farbe, also muss der Assistent eine der Farben auf dem Gerüst auswählen und diese vom Gerüst zurück zum Boden bringen.

Leonardos optimale Strategie

Der Assistent möchte so häufig wie möglich Pause machen. Die Anzahl Anforderungen, für die er nichts machen muss, hängt von seinen Entscheidungen ab, die er während des Tages trifft. Genauer gesagt: Jedes Mal wenn der Assistent eine Farbe vom Gerüst herunterholen muss, können

unterschiedliche Entscheidungen zu unterschiedlichen Auswirkungen in der Zukunft führen. Leonardo erklärt ihm, wie er sein Ziel erreichen kann, wenn er C kennt. Die beste Art, um die optimale vom Gerüst zu entfernende Farbe zu bestimmen, wird durch die Begutachtung der Farben, die sich momentan auf dem Gerüst befinden, und der verbleibenden Farbanforderungen in C bestimmt. Eine Farbe soll mit den folgenden Regeln aus den Farben auf dem Gerüst ausgewählt werden:

- Falls es eine Farbe auf dem Gerüst gibt, die in der Zukunft nie mehr benötigt wird, soll der Assistent eine solche Farbe vom Gerüst herunterholen.
- Andernfalls soll die Farbe, welche vom Gerüst entfernt wird, diejenige sein, *welche am weitesten in der Zukunft das nächste Mal gebraucht wird*. (Das bedeutet, dass wir für jede der Farben auf dem Gerüst das erste zukünftige Vorkommen in den Anforderungen suchen. Die Farbe, welche zurück auf den Boden gestellt wird, ist diejenige, welche zuletzt gebraucht wird.)

Man kann zeigen, dass der Assistent so oft wie möglich Pause machen kann, wenn er Leonardos Strategie befolgt.

Beispiel 1

Sei N = 4, so haben wir 4 Farben (nummeriert von 0 bis 3) und 4 Anforderungen. Sei die Anforderungssequenz C = (2, 0, 3, 0). Nimm an, dass K = 2. Also hat Leonardo ein Gerüst, welches zu jeder Zeit 2 Farben enthalten kann. Wie oben bemerkt enthält das Gerüst am Anfang die Farben 0 und 1. Wir werden den Inhalt des Gerüstes wie folgt beschreiben: [0, 1]. Eine mögliche Art, wie der Assistent die Anforderungen abarbeiten kann, ist wie folgt:

Die erste angeforderte Farbe (Nummer 2) ist nicht auf dem Gerüst. Der Assistent stellt sie aufs Gerüst und beschliesst, Farbe 1 vom Gerüst zu entfernen. Das aktuelle Gerüst ist [0, 2].

- Die nächste angeforderte Farbe (Nummer 0) ist bereits auf dem Gerüst. Der Assistent kann sich also ausruhen.

- Für die dritte Anforderung (Nummer 3) entfernt der Assistent die Farbe 0, was das Gerüst wie folgt abändert: [3, 2].

- Als letztes muss die letzte angefordete Farbe (Nummer 0) vom Boden aufs Gerüst gelegt werden.

Beachte, dass im obigen Beispiel der Assistent nicht Leonardos optimale Strategie befolgt hat. Die optimale Strategie würde die Farbe 2 im dritten Schritt entfernen, sodass der Assistent im letzten Schritt wieder eine Pause machen kann.

Strategie des Assistenten bei limitiertem Gedächtnis

Am Morgen bittet der Assistent Leonardo darum, C auf einem Stück Papier aufzuschreiben, damit er die optimale Strategie herausfinden und befolgen kann. Aber Leonardo legt grossen Wert darauf, seine Arbeitstechniken geheim zu halten, und weigert sich deshalb, dem Assistenten das Papier zu geben. Er erlaubt dem Assistenten lediglich, C zu lesen und zu versuchen sich C zu merken.

Leider ist das Gedächtnis des Assistenten sehr schlecht. Er kann sich nur bis zu M Bits merken. Generell könnte ihn dies vom Rekonstruieren der ganzen Sequenz C abhalten. Folglich muss der Assistent eine clevere Art finden, um diejenige Bitsequenz zu berechnen, welche er sich merken wird. Wir nennen diese Sequenz *Advice-Sequenz* und wir bezeichnen sie mit A.

Beispiel 2

Am Morgen kann der Assistent Leonardos Papier mit der Anforderungsfolge C lesen und alle notwendigen Entscheidungen treffen. Er könnte sich beispielsweise dafür entscheiden, nach jeder Anforderung den Zustand des Gerüstes zu untersuchen. Wenn er zum Beispiel die (suboptimale) Strategie aus Beispiel 1 verwendet, wäre die Sequenz der Gerüstzustände [0, 2], [0, 2], [3, 2], [3, 0]. (Erinnere dich daran, dass er weiss, dass der Anfangszustand des Gerüstes [0, 1] ist.)

Wir nehmen nun an, dass wir M = 16 haben, also dass sich der Assistent 16 Bits merken kann. Da N = 4, können wir jede Farbe mit 2 Bits speichern. Also genügen 16 Bits, um die obige Folge von Zuständen des Gerüstes zu speichern. Darum berechnet der Assistent die folgende Advice-Sequenz: A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`).

Später am Tag kann der Assistent seine Advice-Sequenz dekodieren und für seine Entscheidungen verwenden.

(Mit M = 16 kann sich der Assistent natürlich auch die ganze Sequenz C merken und braucht dafür nur 8 der verfügbaren 16 Bits. In diesem Beispiel wollen wir nur zeigen, dass es auch andere Möglichkeiten gibt, ohne eine gute Lösung zu verraten.)

Aufgabe

Du musst *zwei getrennte Programme* in der selben Programmiersprache schreiben. Diese Programme werden sequenziell ausgeführt, ohne dass sie während der Ausführung miteinander kommunizieren können.

Das erste Programm wird dasjenige sein, welches der Assistent am Morgen benützt. Dieses Programm bekommt die Folge C und muss eine Advice-Sequenz A berechnen.

Das zweite Programm wird dasjenige sein, welches der Assistent während des Tages verwendet. Dieses Programm bekommt die Advice-Sequenz A und muss dann die Folge C von Leonardos Anforderungen abarbeiten. Beachte, dass die Folge C für dieses Programm nur Anforderung für Anforderung offengelegt wird und dass jede Anforderung bearbeitet werden muss, bevor die nächste Anforderung empfangen werden darf.

Genauer gesagt musst du im ersten Programm ein einzelnes Unterprogramm `ComputeAdvice`(C, N, K, M) schreiben, welches als Eingabe den Array C, bestehend aus N Ganzzahlen (jede aus 0, ..., N - 1), die Anzahl Farben auf dem Gerüst K, und die Anzahl Bits M, die für den Advice zur Verfügung stehen, erhält. Dieses Programm muss eine Advice-Sequenz A berechnen, welche aus bis zu M Bits besteht. Das Programm muss dann den Advice A dem System mitteilen, indem für jedes Bit in A der Reihe nach die folgende Funktion aufgerufen wird:

- `WriteAdvice`(B) — hängt das Bit B an die aktuelle Advice-Sequenz A an. (Du kannst diese Funktion maximal M Mal aufrufen.)

Im zweiten Programm musst du ein einzelnes Unterprogramm `Assist`(A, N, K, R) schreiben. Die Eingabe für dieses Unterprogramm ist die Advice-Sequenz A, die Ganzzahlen N und K, definiert wie oben, und die effektive Länge R der Advice-Sequenz A in Bits ($R \leq M$). Dieses Unterprogramm soll deine vorgeschlagene Strategie des Assistenten ausführen unter Verwendung der folgenden Funktionen, welche dir zur Verfügung gestellt werden:

`GetRequest`() — gibt die Farbe zurück, welche Leonardo als nächstes anfordert. (Dabei werden keine Informationen über die zukünftigen Anforderungen offen gelegt).

- `PutBack`(T) — stelle die Farbe T vom Gerüst zurück auf den Boden. Du darfst diese Funktion nur mit einem Argument T aufrufen, falls T eine der Farben ist, die sich momentan auf dem Gerüst befindet.

Dein Unterprogramm `Assist` muss genau N mal die Funktion `GetRequest` aufrufen. Jeder Rückgabewert der Funktion entspricht einer Farbanforderung von Leonardo (die Anforderungen sind in der richtigen Reihenfolge). Nach jedem Aufruf `GetRequest` "muss" die Funktion `PutBack`(T) mit deiner Wahl von T aufgerufen werden, sofern die angeforderte Farbe sich nicht auf dem Gerüst befindet. Andernfalls darf die Funktion `PutBack` *keinesfalls* aufgerufen werden. Ein Verstoß gegen diese Regeln gilt als Fehler und führt zum Abbruch deines Programms. Beachte bitte, dass am Anfang sich die Farben 0 bis K - 1 auf dem Gerüst befinden.

Ein spezifischer Testfall wird als gelöst betrachtet, falls deine Programme alle auferlegten Restriktionen befolgen und die totale Anzahl Aufrufe von `PutBack` *exakt übereinstimmt* mit derjenigen von Leonardos optimaler Strategie. Beachte: falls es mehrere Strategien gibt, welche dieselbe Anzahl Aufrufe von `PutBack` erreichen, darf dein Programm eine beliebige davon ausführen. (D.h., es wird nicht verlangt, dass genau Leonardos Strategie befolgt wird, falls eine ebenso gute Strategie existiert.)

Beispiel 3

Wir setzen Beispiel 2 fort. Nimm an, als Ergebnis von `ComputeAdvice` sei A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Um das dem System zu übermitteln, würde diese Folge von Aufrufen benötigt:

`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)`.

Danach würde dein Unterprogramm `Assist` ausgeführt, mit der obigen Folge A und den Werten N = 4, K = 2 und R = 16 als Parameter. `Assist` muss dann genau N = 4 mal `GetRequest` aufrufen, sowie nach einigen dieser Anforderungen auch `PutBack`(T) mit einem geeigneten Wert für T.

Die folgende Tabelle zeigt eine Abfolge von Aufrufen gemäß dem (suboptimalen) Vorgehen von Beispiel 1. Der Strich bedeutet, dass `PutBack` nicht aufgerufen wird.

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

Subtask 1 [8 Punkte]

- $N \leq 5\,000$.
- Du darfst maximal M = 65 000 Bits verwenden.

Subtask 2 [9 Punkte]

- $N \leq 100\,000$.
- Du darfst maximal M = 2 000 000 Bits verwenden.

Subtask 3 [9 Punkte]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Du darfst maximal M = 1 500 000 Bits verwenden.

Subtask 4 [35 Punkte]

- $N \leq 5\,000$.
- Du darfst maximal M = 10 000 Bits verwenden.

Subtask 5 [bis zu 39 Punkte]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Du darfst maximal M = 1 800 000 Bits verwenden.

Deine Punktzahl für diesen Subtask hängt von der Länge R des "Advice" ab, den dein Programm benötigt. Genauer gesagt: wenn R_{max} die maximale Länge (über alle Testfälle dieses Subtasks) des "Advice" ist, den dein Programm in der Funktion `ComputeAdvice` produziert, ist deine Punktzahl:

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

Implementierungsdetails

Du musst genau zwei Dateien mit Programmen abgeben, *die alle in der gleichen Sprache geschrieben sind*.

Die erste Datei heißt `advisor.c`, `advisor.cpp` oder `advisor.pas`. Sie muss das Unterprogramm `ComputeAdvice` implementieren, wie oben beschrieben, und kann das Unterprogramm `WriteAdvice` aufrufen. Die zweite Datei heißt `assistant.c`, `assistant.cpp` oder `assistant.pas`. Sie muss das Unterprogramm `Assist` implementieren, wie oben beschrieben, und kann die Unterprogramme `GetRequest` und `PutBack` aufrufen.

Es folgen die Signaturen aller Unterprogramme.

C/C++ Programme

```
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 Programme

```
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;
```

Diese Unterprogramme müssen sich wie oben beschrieben verhalten. Du darfst natürlich weitere Unterprogramme zur internen Verwendung schreiben. In C/C++ Programmen sollten deine Unterprogramme als static deklariert sein, denn der Beispiel-Grader wird sie zusammenlinken. Alternativ muss man darauf verzichten, einen Funktionennamen in beiden Quelltextdateien gleichzeitig zu verwenden. Deine Einsendungen dürfen in keinster Weise mit Standard Input, Standard Output oder irgendeiner Datei interagieren.

Beachte beim Programmieren auch die folgenden Anweisungen; die Header-Dateien auf deinem PC genügen den nun genannten Bedingungen bereits.

C/C++ Programme

Am Anfang deiner Programme musst du die Datei `advisor.h` bzw. `assistant.h` einbinden, in den "advisor" bzw. den "assistant", und zwar durch die folgende Zeile:

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

oder

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

Die Dateien `advisor.h` und `assistant.h` stehen dir in einem Verzeichnis auf deinem PC zur Verfügung und werden auch über das Webinterface angeboten. Ebenso werden dir Quelltext und Skripte zum Kompilieren und Testen deiner Lösung angeboten. Nachdem du deine Lösung in das Verzeichnis mit diesen Skripten kopiert hast, musst du dazu `compile_c.sh` bzw. `compile_cpp.sh` ausführen.

Pascal-Programme

Du musst die Units `advisorlib` und `assistantlib` für den "advisor" bzw. den "assistant" verwenden. Dazu musst du in deinem Quelltext die folgende Zeile einfügen:

```
uses advisorlib;
```

bzw.

```
uses assistantlib;
```

Die zwei Dateien `advisorlib.pas` und `assistantlib.pas` stehen dir in einem Verzeichnis auf deinem PC zur Verfügung und werden auch über das Webinterface angeboten. Ebenso werden dir Quelltext und Skripte zum Kompilieren und Testen deiner Lösung angeboten. Nachdem du deine Lösung in das Verzeichnis mit diesen Skripten kopiert hast, musst du dazu `compile_pas.sh` ausführen.

Beispiel-Grader

Der Beispiel-Grader liest die Eingabe im folgenden Format:

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

Der Beispiel-Grader führt zunächst das Unterprogramm `ComputeAdvice` aus. Er generiert eine Datei `advice.txt`, die die einzelnen Bits des "advice" durch Leerzeichen voneinander getrennt enthält; am Ende der Datei steht eine einzelne 2.

Anschließend führt es dein `Assist` Unterprogramm aus, und produziert eine Ausgabe, in der jede Zeile entweder von der Form "`R [number]`" oder von der Form "`P [number]`" ist. Jede

Zeile, die mit R beginnt, gibt das Ergebnis eines Aufrufs von `GetRequest()` an. Jede Zeile, die ein P enthält, entspricht einem Aufruf von `PutBack`() und gibt die Farbe, die abgestellt wurde, an. Die Ausgabe wird durch eine Zeile der Form "`E`" abgeschlossen.

Beachte, dass bei Verwendung des offiziellen Graders die Laufzeit deines Programms leicht anders ist. Die Differenz sollte nicht signifikant sein. Du darfst das Testinterface verwenden, um deinen Code mit dem offiziellen Grader laufen zu lassen.

Limits

- Zeitlimit: 7 Sekunden.
- Speicherlimit: 256 MiB.