Het Laatste Avondmaal

Leonardo was lang bezig aan zijn meest bekende muurschildering: het Laatste Avondmaal. Eén van de eerste dingen die hij 's ochtends moest doen, was selecteren welke kleuren verf hij die dag nodig zou hebben. Dat waren er veel, maar hij kon er maar enkele bij zich houden op de steiger waarop hij aan het schilderij werkte. Het was de taak van zijn leerling om de steiger op te klimmen om hem kleuren te bezorgen, en zonodig kleuren naar beneden te halen en terug in de kast te zetten.

In deze taak moet je twee aparte programma's schrijven om de leerling te helpen. Het eerste programma ontvangt Leonardo's instructies (een reeks kleuren die Leonardo die dag nodig zal hebben), en produceert een *korte* reeks bits die we *advice* zullen noemen. Terwijl de de leerling de opdrachten van Leonardo uitvoert, zal de leerling geen toegang hebben tot Leonardo's daaropvolgende opdrachten, enkel tot het advice geproduceerd door je eerste programma. Het tweede programma krijgt dat advice, en zal op een *online* manier (d.w.z. één voor één) de instructies van Leonardo ontvangen en verwerken. Dit programma moet kunnen begrijpen wat het advice betekent, en het gebruiken om optimale keuzes te maken. Dit wordt later in meer detail uitgelegd.

Kleuren transporteren tussen kast en steiger

Beschouw dit vereenvoudigd scenario. Stel dat er N kleuren zijn, genummerd van 0 tot en met N-1, en dat Leonardo zijn leerling iedere dag ook N keer om een nieuwe kleur verzoekt. Laat C de reeks zijn van de N kleurverzoeken van Leonardo. Dan kunnen we C beschouwen als een reeks van N getallen met mogelijke waarden van 0 tot en met N - 1. Merk op dat sommige kleuren misschien niet in C voorkomen, terwijl andere kleuren meermaals aanwezig kunnen zijn.

De steiger staat altijd vol. Ze kan een bepaald aantal K van de N kleuren bevatten, met $K < N$. In het begin staan steeds de kleuren 0 tot en met K - 1 op de steiger.

De leerling voert steeds één opdracht van Leonardo tegelijk uit. Wanneer een gevraagde kleur *al op de steiger aanwezig* is, kan de leerling uitrusten. Anders moet hij de gevraagde kleur uit de kast halen en naar boven brengen. Uiteraard is daar geen plaats voor de nieuwe kleur, dus de leerling moet één van de daar aanwezige kleuren kiezen om van de steiger af te halen en terug in de kast te zetten.

Leonardo's optimale strategie

De leerling wil zo vaak mogelijk uitrusten, maar het aantal keren dat hij kan rusten hangt af van de keuzes die hij maakt. Meer bepaald: elke keer dat de leerling een kleur moet kiezen om van de steiger te halen, kan een andere keuze leiden tot een ander resultaat in de toekomst. Leonardo legt

hem uit hoe hij zijn doel kan bereiken als hij C kent. De beste keuze voor een kleur die moet worden weggehaald kan worden bepaald door te bekijken welke kleuren momenteel op de steiger staan, en welke kleurverzoeken er nog overblijven in C. Van de kleuren op de steiger zou er één gekozen moeten worden op basis van de volgende regels:

- Als er een kleur op de steiger staat die nooit meer nodig is in de toekomst dient de leerling die van de steiger te halen.
- Anders dient de kleur verwijderd te worden *die pas het verst in de toekomst terug nodig zal zijn*. (Dat wil zeggen, voor elk van de kleuren op de steiger vinden we de eerstvolgende toekomstige vraag ernaar. De kleur die moet weggehaald worden is diegene die het laatst terug nodig is.)

Het kan worden bewezen dat de leerling maximaal kan uitrusten wanneer hij Leonardo's strategie gebruikt.

Voorbeeld 1

Stel N = 4, dus we hebben 4 kleuren (genummerd van 0 tot 3) en 4 kleurverzoeken. Stel de reeks verzoeken is C = (2, 0, 3, 0). Stel ook K = 2, dus Leonardo kan 2 kleuren tegelijk op zijn steiger bewaren. Zoals hierboven uitgelegd bevat de steiger in het begin kleuren 0 en 1. We beschrijven de inhoud van de steiger als [0, 1]. Eén van de manieren waarop de leerling de verzoeken kan verwerken is als volgt.

- De eerste gevraagde kleur (nummer 2) staat niet op de steiger. De leerling brengt die naar boven en verwijdert kleur 1 van de steiger. De steiger bevat nu: [0, 2].

- De volgende gevraagde kleur (nummer 0) is al aanwezig op de steiger, dus de leerling kan rusten.

- Voor het derde verzoek (nummer 3) haalt de leerling kleur 0 weg. De steiger bevat nu: [3, 2].

- Tenslotte wordt kleur 0 gevraagd en naar boven gebracht. De leerling neem kleur 2 weg en de steiger bevat nu: [3, 0].

Merk op dat de leerling in dit voorbeeld niet Leonardo's ideale strategie heeft gebruikt. De optimale strategie zou zijn om kleur 2 weg te halen in de derde stap, zodat hij bij het laatste verzoek opnieuw kon rusten.

De strategie van de leerling met een klein geheugen

De leerling vraagt Leonardo 's ochtends om C op papier te schrijven zodat hij de optimale strategie kan vinden en volgen. Leonardo staat er echter op om zijn werkmethode geheim te houden, en dus laat hij niet toe dat de leerling dit papier bijhoudt. In plaats daarvan moet de leerling C lezen en proberen te onthouden.

De leerling heeft helaas een slecht geheugen. Hij kan maximaal M bits onthouden. In het algemeen kan hij daardoor niet de hele reeks C reconstrueren. Daarom moet de leerling op een slimme manier een reeks bits berekenen die hij wel kan onthouden. Die reeks noemen we de *advice-reeks*, en we noteren die als A.

Voorbeeld 2

's Morgens kan de leerling Leonardo's papier met reeks C pakken, lezen en de nodige keuzes maken. Hij zou bijvoorbeeld kunnen kiezen om de staat van de steiger te bekijken na elk van de verzoeken. Wanneer hij de (niet optimale) strategie van voorbeeld 1 gebruikt, geeft dat de reeks [0, 2], [0, 2], [3, 2], [3, 0]. (Merk op dat hij weet dat de eerste toestand van de steiger [0, 1] is.)

Stel nu dat M = 16, dus de leerling kan 16 bits aan informatie onthouden. Gezien N = 4 kunnen we elke kleur opslaan in 2 bits. Dan zijn 16 voldoende om de reeks van steigertoestanden te onthouden. De leerling kan de volgende advice-reeks berekenen: A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`).

Later op de dag kan de leerling deze advice-reeks decoderen en op basis hiervan zijn keuzes maken.

(Uiteraard kan de leerling met M = 16 ook de hele reeks C coderen en onthouden, door slechts 8 van de 16 bits te gebruiken. In dit voorbeeld willen we alleen aangeven dat hij meerdere mogelijkheden heeft, zonder hiermee een goede oplossing van het probleem te verklappen.)

Opdracht

Je moet *twee aparte programma's* schrijven in dezelfde programmeertaal. Deze programma's worden na elkaar uitgevoerd, en kunnen niet met elkaar communiceren tijdens het uitvoeren.

Het eerste programma wordt door de leerling 's ochtends gebruikt. Dit programma krijgt als invoer de reeks C en moet een advice-reeks A berekenen.

Het tweede programma wordt door de leerling later die dag gebruikt. Dit programma krijgt de advice-reeks A en moet hierna de reeks C met met Leonardo's verzoeken verwerken. Let op dat de reeks C verzoek per verzoek wordt kenbaar gemaakt aan je programma, en dat elk verzoek verwerkt moet worden voordat het volgende gegeven wordt.

Meer bepaald: in het eerste programma moet je de functie `ComputeAdvice`(C, N, K, M) implementeren die als input krijgt: de array C van N integers (met mogelijke waarden van 0 tot en met N - 1), het aantal kleuren K op de steiger, en het aantal bits M dat beschikbaar is voor het advice. Dit programma moet een advice-reeks A berekenen die maximaal uit M bits mag bestaan. Het programma moet vervolgens de reeks A aan het systeem communiceren door bit voor bit de volgende functie aan te roepen:

- `WriteAdvice`(B) — voegt bit B toe aan de al eerder doorgegeven advice-reeks A. (Je mag deze functie maximaal M keer aanroepen.)

In het tweede programma implementeer je de functie `Assist`(A, N, K, R). De input voor deze functie is de advice-reeks A, de integers N en K zoals hierboven gedefinieerd, en de lengte R van de reeks A in bits ($R \leq M$). Deze functie moet jouw strategie uitvoeren voor de leerling, en je krijgt daarvoor de volgende functies:

- `GetRequest`() — geeft het volgende kleurverzoek van Leonardo terug. (De rest van de verzoeken krijg je pas later.)

- `PutBack`(T) — haalt de kleur T van de steiger terug naar de kast. Je mag deze functie enkel aanroepen als T daadwerkelijk op de steiger staat.

Jouw functie `Assist` moet `GetRequest` precies N keer aanroepen, telkens ontvang je het volgende kleurverzoek van Leonardo. Na iedere aanroep van `GetRequest` die een kleur teruggeeft die *niet* op de steiger staat, *moet* je de functie `PutBack`(T) aanroepen met jouw keuze voor weg te halen kleur T. Anders mag je `PutBack` *niet* aanroepen. Als je je hier niet aan houdt wordt dat beschouwd als fout en wordt je programma beëindigd. Nogmaals, denk er aan dat de steiger in het begin de kleuren 0 tot en met K - 1 bevat.

Een specifieke testcase wordt als opgelost beschouwd als jouw twee functies aan alle opgelegde voorwaarden voldoen, en het aantal aanroepen van `PutBack` *exact gelijk* is aan dat van Leonardo's optimale strategie. Let op: Als er verschillende strategieën bestaan die resulteren in hetzelfde aantal aanroepen van `PutBack`, dan mag jouw programma eender welke ervan gebruiken. (Het is dus niet noodzakelijk om precies de strategie van Leonardo te volgen, als er een andere strategie bestaat die net zo goed is).

Voorbeeld 3

We gaan verder met voorbeeld 2. Stel dat je in `ComputeAdvice` deze advice-reeks berekent: A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Om deze aan het systeem te communiceren moet je de volgende reeks functie-aanroepen maken:

`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`).

Jouw tweede functie `Assist` wordt dan uitgevoerd, die de bovengenoemde reeks A ontvangt, alsook de waarden N = 4, K = 2, en R = 16. De functie `Assist` moet dan precies N = 4 keer de functie `GetRequest` aanroepen. Na sommige van deze aanroepen zal `Assist` ook `PutBack`(T) moeten aanroepen met een geschikte keuze voor T.

De tabel hieronder geeft een reeks aanroepen weer die overeenkomen met de (niet optimale) keuzes van voorbeeld 1. Het streepje geeft aan dat `PutBack` niet wordt aangeroepen.

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

Subtaak 1 [8 punten]

- $N \leq 5\,000$.
- Je mag maximaal M = 65 000 bits gebruiken.

Subtaak 2 [9 punten]

- $N \leq 100\,000$.
- Je mag maximaal M = 2 000 000 bits gebruiken.

Subtaak 3 [9 punten]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Je mag maximaal M = 1 500 000 bits gebruiken.

Subtaak 4 [35 punten]

- $N \leq 5\,000$.
- Je mag maximaal M = 10 000 bits gebruiken.

Subtaak 5 [maximaal 39 punten]

- $N \leq 100\,000$.
- $K \leq 25\,000$.
- Je mag maximaal M = 1 800 000 bits gebruiken.

De score voor deze subtaak hangt af van de lengte R van het advice dat je programma naar het systeem communiceert. Meer bepaald, als R_{max} het maximum is (over alle testcases) van de lengte van de advice-reeks die door jouw functie `ComputeAdvice` wordt geproduceerd, wordt je score:

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

Implementatiedetails

Je moet precies twee bestanden indienen *in dezelfde programmeertaal*.

Het eerste bestand heet `advisor.c`, `advisor.cpp` of `advisor.pas`. Dit bestand moet de

functie `ComputeAdvice` implementeren, zoals hierboven beschreven. Het kan de functie `WriteAdvice` aanroepen. Het tweede bestand heet `assistant.c`, `assistant.cpp` of `assistant.pas`. Dit bestand moet de functie `Assist` implementeren, zoals hierboven beschreven. Het kan de functies `GetRequest` en `PutBack` aanroepen.

De declaraties van alle functies zijn als volgt:

C/C++ programma's

```
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 programma's

```
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;
```

Deze functies moeten werken zoals hierboven werd beschreven. Natuurlijk staat het je vrij bijkomende functies te implementeren voor intern gebruik. In C/C++ programma's moet je zulke interne functies declareren als `static`, omdat de voorbeeld-grader ze zal linken. Vermijd om twee functies dezelfde naam te geven, al zitten ze in verschillende bestanden. Je code mag op geen enkele manier interageren met standaard input/output, of met andere bestanden.

Let bij het programmeren ook op de volgende instructies (de skeletcode die je kan vinden in de wedstrijdomgeving voldoet al aan de hieronder genoemde eisen).

C/C++ programma's

Aan het begin van je oplossing moet je in jouw *advisor* en jouw *assistant* code respectievelijk de bestanden `advisor.h` en `assistant.h` includen. Dat doe je door de volgende lijn toe te voegen:

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

of

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

De bestanden `advisor.h` en `assistant.h` krijg je, ze staan in een map in je wedstrijdomgeving. Je kunt ze ook op de Web-interface van de wedstrijd vinden. Op deze manier

zal je ook code en scripts krijgen om je oplossing te compileren en testen. Dat doe je door je oplossing in de directory met deze scripts te kopiëren, en het script `compile_c.sh` of `compile_cpp.sh` uit te voeren (afhankelijk van de keuze van programmeertaal).

Pascal programma's

Gebruik de units `advisorlib` en `assistantlib`, voor respectievelijk de advisor en de assistant. Dit doe je door de volgende regel in je code te zetten:

```
uses advisorlib;
```

of

```
uses assistantlib;
```

De bestanden `advisorlib.pas` en `assistantlib.pas` krijg je, ze staan in een map in je wedstrijdomgeving. Je kunt ze ook op de Web-interface van de wedstrijd vinden. Op deze manier zal je ook code en scripts krijgen om je oplossing te compileren en testen. Dat doe je door je oplossing in de directory met deze scripts te kopiëren, en het script `compile_pas.sh` uit te voeren

Voorbeeld-grader

De voorbeeld-grader verwacht invoer in het volgende formaat:

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

De grader voert eerst de functie `ComputeAdvice` uit. Dit genereert een bestand `advice.txt` dat de individuele bits van de advice-reeks bevat, gescheiden door spaties en afgesloten met een 2.

Daarna wordt jouw functie `Assist` uitgevoerd. Daarbij wordt uitvoer gegenereerd waarbij elke regel ofwel van de vorm "`R [getal]`", ofwel van de vorm "`P [getal]`" is. Regels van de eerste vorm duiden op een aanroep van `GetRequest()`, met daarachter het ontvangen antwoord. Regels van de tweede vorm geven duiden op een aanroep van `PutBack`() en de gekozen kleur om weg te halen. De uitvoer wordt afgesloten met een regel die eruit ziet als "`E`".

Let op dat de runtime van je programma op de officiële grader kan afwijken van die op je eigen computer. Dit verschil is normaalgezien niet significant. Toch raden we je aan om de test interface te gebruiken om te controleren of je oplossing binnen de tijdslimiet blijft.

Tijds- en geheugenlimieten

- Tijdslimiet: 7 seconden.
- Geheugenlimiet: 256 MiB.