Ostatnia Wieczerza

Gdy Leonardo da Vinci malował swój najsłynniejszy fresk, Ostatnią Wieczerzę, pracował całymi dniami. Każdego dnia, wczesnym rankiem zastanawiał się, których kolorów farb będzie w ciągu tego dnia potrzebować. Zazwyczaj używał on wielu barw, jednak pod ręką (tj. na rusztowaniu) mógł mieć jedynie ograniczoną liczbę pojemników z farbą. Wnoszeniem pojemników z farbą na rusztowanie i znoszeniem ich na półkę stojącą na podłodze zajmował się asystent Leonarda.

Twoim zadaniem będzie napisanie dwóch osobnych programów, które pomogą asystentowi. Pierwszy program otrzyma opis zamiarów Leonarda (ciąg kolorów farb, których będzie kolejno potrzebować malarz w ciągu dnia) i obliczy *krótki* ciąg bitów, który nazywać będziemy *podpowiedzią*. Podczas obsługiwania próśb Leonarda o pojemniki z farbą, asystent nie będzie znał jego przyszłych żądań, a jedynie podpowiedź obliczoną przez Twój program. Drugi program otrzyma podpowiedź, a następnie otrzymywać będzie żądania Leonarda *online* (tj. jedno na raz). Program będzie musiał zrozumieć podpowiedź i użyć jej do optymalnego przenoszenia pojemników z farbą. Jeśli chcesz poznać szczegóły, czytaj dalej.

Przenoszenie pojemników między półką a rusztowaniem

Rozważymy nieco uproszczoną sytuację. Załóżmy, że Leonardo dysponuje N pojemnikami z farbą ponumerowanymi od 0 do N - 1 (każdy pojemnik zawiera farbę innego koloru) i każdego dnia dokładnie N razy prosi asystenta o pewien pojemnik. Niech C oznacza ciąg opisujący żądania Leonarda. Ciąg ten ma długość N, a jego wyrazy to liczby z zakresu od 0 do N - 1. Może się zdarzyć, że pewne liczby będą występować w ciągu C wiele razy; może się również zdarzyć, że pewne liczby w ogóle w nim nie wystąpią.

Rusztowanie jest przez cały czas w pełni zastawione pojemnikami z farbą: w każdej chwili znajduje się na nim K (spośród N) pojemników, przy czym $K < N$. Początkowo rusztowanie zawiera pojemniki o numerach od 0 do K - 1.

Asystent obsługuje żądania Leonarda po kolei. Jeśli Leonardo prosi o pojemnik z farbą, który aktualnie znajduje się na rusztowaniu, asystent nie musi nic robić i odpoczywa. W przeciwnym razie powinien on wziąć pojemnik z farbą żądanego koloru z półki i przenieść go na rusztowanie. Oczywiście na rusztowaniu nie ma miejsca na nowy pojemnik, więc asystent musi wybrać jeden z pojemników z rusztowania i znieść go na półkę.

Optymalna strategia noszenia pojemników

Asystent chciałby się jak najmniej napracować. Oczywiście liczba żądań, które spowodują, że będzie on musiał nosić pojemniki, zależy od jego wyborów. Dzieje się tak, gdyż za każdym razem, gdy asystent musi zdecydować, który pojemnik powinien znieść z rusztowania na półkę, jego

decyzja wpływa na przebieg przyszłych wydarzeń. Leonardo wie, jak powinien postępować asystent (jeśli zna ciąg C), by mieć jak najmniej pracy. Przy podejmowaniu decyzji powinien zwrócić uwagę na pojemniki aktualnie stojące na rusztowaniu oraz żądania, które w przyszłości wystosuje Leonardo, a następnie wybrać pojemnik w następujący sposób:

- Jeśli na rusztowaniu znajduje się pojemnik, który nie będzie potrzebny w przyszłości, asystent powinien znieść go na półkę.
- W przeciwnym razie, znieść należy pojemnik, o który Leonardo poprosi jak najpóźniej. Innymi słowy, dla każdego pojemnika na półce znajdujemy jego następne wystąpienie w ciągu żądań Leonarda. Na półkę należy odnieść pojemnik, którego następne wystąpienie jest najpóźniejsze.

Można wykazać, że postępując zgodnie ze strategią Leonarda, asystent będzie odpoczywał maksymalną możliwą liczbę razy.

Przykład 1

Załóżmy, że N = 4. Mamy wówczas 4 kolory (ponumerowane od 0 do 3) oraz 4 żądania. Przyjmijmy, że ciąg żądań to C = (2, 0, 3, 0). Ponadto załóżmy, że K = 2. Oznacza to, że na rusztowaniu mieszczą się 2 pojemniki z farbą. Początkowo rusztowanie zawiera pojemniki o numerach 0 i 1 (mówimy, że zawartość rusztowania to [0, 1]). Asystent może wówczas obsłużyć żądania Leonarda na przykład tak:

- Pierwszego pojemnika, o który prosi Leonardo (numer 2), nie ma na rusztowaniu. Asystent wnosi więc pojemnik na rusztowanie i postanawia znieść pojemnik 1 na półkę. Zawartość rusztowania to [0, 2].

- Następny pojemnik (numer 0) znajduje się na rusztowaniu, więc asystent może odpocząć.

- Przy trzecim żądaniu (pojemnik 3) asystent zdejmuje pojemnik 0 na półkę; nowa zawartość rusztowania to [3, 2].

- Ostatni pojemnik, o który prosi Leonardo (numer 0), musi zostać przyniesiony z półki. Asystent postanawia zdjąć pojemnik 2 i w efekcie zawartość rusztowania to [3, 0].

Zwróć uwagę, że w powyższym przykładzie asystent nie stosował optymalnej strategii podanej przez Leonarda. Przy optymalnej strategii asystent powinien bowiem w trzecim kroku usunąć pojemnik numer 2 z rusztowania, co pozwoliłoby mu odpocząć przy ostatnim żądaniu.

Strategia asystenta przy ograniczonej pamięci

Wczesnym rankiem asystent poprosił Leonarda o zapisanie ciągu C na kartce, żeby móc zawczasu znaleźć optymalną strategię działania. Jednak Leonardo zazdrośnie strzeże sekretów swojej pracy i nie zgodził się przekazać asystentowi ciągu C na piśmie. Pozwolił mu jedynie przeczytać ciąg C; asystent może więc podjąć próbę jego zapamiętania.

Niestety pamięć asystenta potrafi zmieścić tylko M bitów informacji. Może to oznaczać, że asystent nie będzie mógł zapamiętać całego ciągu C. Asystent musi zatem wymyślić sprytny sposób na zapamiętanie pewnych informacji w M bitach. Zapamiętany przez niego ciąg bitów będziemy nazywać *ciągiem podpowiedzi* i oznaczymy przez A.

Przykład 2

Rankiem asystent może przeczytać ciąg C z notatek Leonarda i stworzyć plan działania. Na przykład, asystent może wyznaczyć zawartość rusztowania po każdym żądaniu. Jeśli, na przykład, postanowiłby użyć (nieoptymalnej) strategii z Przykładu 1, musiałby zapamiętać następujące zawartości rusztowania: [0, 2], [0, 2], [3, 2], [3, 0]. Zauważ, że jasne jest, iż początkowa zawartość rusztowania to [0, 1], i nie trzeba tej informacji zapamiętywać.

Przyjmijmy teraz, że M = 16, czyli asystent potrafi zapamiętać 16 bitów informacji. Skoro N = 4, każdy kolor da się zapisać przy użyciu 2 bitów. Zatem 16 bitów wystarcza do zapisania zawartości rusztowania. Asystent obliczy więc następujący ciąg podpowiedzi: A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`).

W ciągu dnia asystent może odkodować ciąg podpowiedzi i użyć go do podejmowania decyzji.

(Oczywiście, skoro M = 16, asystent mógłby po prostu zapamiętać cały ciąg C przy użyciu zaledwie 8 bitów, jednak celem tego przykładu jest pokazanie, że asystent ma także inne możliwości.)

Zadanie

Twoim zadaniem jest napisanie *dwóch programów* w tym samym języku programowania. Programy te zostaną uruchomione osobno (jeden po drugim) i nie będą mogły się ze sobą komunikować w trakcie wykonania.

Pierwszy program symuluje poranne zachowanie asystenta. Program ten otrzyma ciąg C i powinien obliczyć ciąg podpowiedzi A.

Drugi program powinien symulować zachowanie asystenta w ciągu dnia. Program ten otrzyma ciąg podpowiedzi A (obliczony przez pierwszy program) i powinien obsłużyć ciąg C, opisujący żądania Leonarda. Ciąg C będzie odsłaniany drugiemu programowi wyraz po wyrazie i każde żądanie należy obsłużyć przed poznaniem kolejnego żądania.

Mówiąc dokładniej, w pierwszym programie należy zaimplementować jedną funkcję `ComputeAdvice`(C, N, K, M), która jako argument otrzymuje ciąg C (składający się z N liczb całkowitych z przedziału [0, N-1]), liczbę K oznaczającą liczbę pojemników, które mieszczą się na rusztowaniu, oraz liczbę M oznaczającą, ile bitów potrafi zapamiętać asystent. Program powinien obliczyć ciąg podpowiedzi A, który składa się z co najwyżej M bitów, a następnie podać zawartość tego ciągu przez wywołanie poniższej procedury dla kolejnych bitów:

- `WriteAdvice`(B) — dopisz bit B na końcu aktualnego ciągu podpowiedzi A. Możesz wywołać tę funkcję co najwyżej M razy.

W drugim programie powinieneś zaimplementować funkcję `Assist`(A, N, K, R). Otrzymuje ona ciąg podpowiedzi A, liczby całkowite N i K opisane powyżej oraz liczbę R, która oznacza długość ciągu A w bitach (R ≤ M). Funkcja `Assist` powinna zrealizować strategię asystenta przy użyciu następujących funkcji:

- `GetRequest`() — zwraca numer następnego pojemnika, o który prosi Leonardo. (Nie dostarcza ona jednak informacji o przyszłych żądaniach Leonarda).
- `PutBack`(T) — odłóż pojemnik T z rusztowania na półkę. T musi być numerem jednego z pojemników, które aktualnie znajdują się na rusztowaniu.

Funkcja `Assist` powinna wywołać `GetRequest` dokładnie N razy, za każdym razem otrzymując kolejne żądanie Leonardo. Po każdym wywołaniu `GetRequest`, jeśli żądany pojemnik *nie* znajduje się na rusztowaniu, *należy koniecznie* wywołać `PutBack`(T) dla pewnego T. W przeciwnym razie *nie należy* wywoływać `PutBack`. Niestosowanie się do tego wymagania powoduje natychmiastowe zakończenie programu. Pamiętaj, że na początku na rusztowaniu znajdują się pojemniki o numerach od 0 do K - 1.

Dany test uznaje się za zaliczony, jeśli obydwie napisane przez Ciebie funkcje stosują się do podanych ograniczeń, a liczba wywołań `PutBack` jest *taka sama*, jak w optymalnej strategii Leonarda. Jeśli istnieje wiele strategii o takiej samej liczbie wywołań `PutBack`, jak w optymalnej strategii Leonarda, Twój program może użyć dowolnej z nich. (W szczególności, nie trzeba używać strategii Leonarda, jeśli istnieje inna, równie dobra strategia.)

Przykład 3

Wróćmy do Przykładu 2 i załóżmy, że funkcja `ComputeAdvice` wyznaczyła ciąg A = (`0`, `0`, `1`, `0`, `0`, `0`, `1`, `0`, `1`, `1`, `1`, `0`, `1`, `1`, `0`, `0`). Aby zwrócić ten ciąg, należy wywołać `WriteAdvice`(`0`), `WriteAdvice`(`0`), `WriteAdvice`(`1`), `WriteAdvice`(`0`), `WriteAdvice`(`0`), `WriteAdvice`(`0`), `WriteAdvice`(`1`), `WriteAdvice`(`0`), `WriteAdvice`(`0`), `WriteAdvice`(`1`), `WriteAdvice`(`1`), `WriteAdvice`(`1`), `WriteAdvice`(`0`), `WriteAdvice`(`1`), `WriteAdvice`(`1`), `WriteAdvice`(`0`), `WriteAdvice`(`0`).

Następnie wywołana zostaje funkcja `Assist`. Jej parametry to powyższy ciąg A oraz N = 4, K = 2 i R = 16. Funkcja `Assist` musi N = 4 razy wywołać funkcję `GetRequest`. Ponadto po niektórych wywołaniach `GetRequest`, funkcja `Assist` powinna wywołać również funkcję `PutBack`(T) dla odpowiednio dobranej wartości T.

Poniższa tabelka pokazuje ciąg wywołań odpowiadający (nieoptymalnym) wyborom z Przykładu 1. Kreska oznacza, że `PutBack` nie jest wywoływana.

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

Podzadanie 1 [8 punktów]

- $N \leq 5\,000$
- Można zapamiętać co najwyżej M = 65 000 bitów.

Podzadanie 2 [9 punktów]

- $N \leq 100\,000$
- Można zapamiętać co najwyżej M = 2 000 000 bitów.

Podzadanie 3 [9 punktów]

- $N \leq 100\,000$
- $K \leq 25\,000$
- Można zapamiętać co najwyżej M = 1 500 000 bitów.

Podzadanie 4 [35 punktów]

- $N \leq 5\,000$
- Można zapamiętać co najwyżej M = 10 000 bitów.

Podzadanie 5 [do 39 punktów]

- $N \leq 100\,000$
- $K \leq 25\,000$
- Można zapamiętać co najwyżej M = 1 800 000 bitów.

Punktacja za to podzadanie zależy od długości ciągu podpowiedzi, który wyznaczy Twój program. Niech R_{max} oznacza maksymalną (po wszystkich testach) długość ciągu podpowiedzi obliczoną przez `ComputeAdvice`. Otrzymasz:

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

Szczegóły implementacji

Powinieneś zgłosić dokładnie dwa pliki *w tym samym języku programowania*.

Pierwszy plik powinien nazywać się `advisor.c`, `advisor.cpp` lub `advisor.pas`. Musi on zawierać implementację funkcji `ComputeAdvice` zgodną z powyższym opisem. W implementacji można wywoływać funkcję `WriteAdvice`. Drugi plik powinien nazywać się `assistant.c`, `assistant.cpp` lub `assistant.pas`. Musi on zawierać implementację funkcji `Assist` zgodną z powyższym opisem. W implementacji można wywoływać funkcje

`GetRequest` i `PutBack`.

Funkcje powinny mieć następujące sygnatury.

Programy w 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();
```

Programy w Pascalu

```
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;
```

Funkcje powinny zachowywać się zgodnie z powyższym opisem. Jeśli w obydwóch swoich programach zechcesz zaimplementować jakieś dwie funkcje, które będą miały taką samą nazwę, ich deklaracje musisz poprzedzić słowem kluczowym `static`. Programy nie powinny korzystać ze standardowego wejścia lub wyjścia ani z jakichkolwiek plików.

Twoje rozwiązanie powinno także spełniać poniższe wymagania (w szczególności, spełniają je szablony rozwiązań na Twoim komputerze).

Programy w C/C++

Na początku Twojego rozwiązania należy dołączyć plik `advisor.h` (w programie obliczającym ciąg podpowiedzi) i `assistant.h` (w programie symulującym przenoszenie pojemników) przy pomocy poniższych dyrektyw:

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

lub

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

Pliki `advisor.h` i `assistant.h` znajdują się w katalogu na dysku Twojego komputera. Można je również znaleźć na stronie systemu sprawdzającego. Dostępny jest również (w ten sam sposób) kod i skrypty kompilujące i testujące rozwiązania. Po skopiowaniu Twojego rozwiązania do katalogu z tymi skryptami wystarczy uruchomić `compile_c.sh` lub `compile_cpp.sh` (w zależności od języka programowania).

Programy w Pascalu

Powinieneś użyć modułów `advisorlib` (w programie obliczającym ciąg podpowiedzi) i `assistantlib` (w programie symulującym przenoszenie pojemników). W tym celu, plik z rozwiązaniem powinien zawierać:

```
uses advisorlib;
```

lub

```
uses assistantlib;
```

Pliki `advisorlib.pas` i `assistantlib.pas` znajdują się w katalogu na dysku Twojego komputera. Można je również znaleźć na stronie systemu sprawdzającego. Dostępny jest również (w ten sam sposób) kod i skrypty kompilujące i testujące rozwiązania. Po skopiowaniu Twojego rozwiązania do katalogu z tymi skryptami wystarczy uruchomić `compile_pas.sh`.

Przykładowy moduł oceniający

Przykładowy moduł oceniający wczytuje dane w poniższym formacie:

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

Przykładowy moduł oceniający na początku wywoła funkcję `ComputeAdvice`, co spowoduje utworzenie pliku `advice.txt`, który zawierać będzie poszczególne bity ciągu podpowiedzi, pooddzielane pojedynczymi odstępami. Ciąg będzie zakończony liczbą 2.

Obliczony ciąg zostanie następnie przekazany Twojej implementacji funkcji `Assist`. Na wyjście zostanie wypisana pewna liczba wierszy. Każdy z tych wierszy zawierać będzie albo "`R [liczba]`", albo "`P [liczba]`". Wiersze pierwszego typu oznaczają wywołania `GetRequest()` i otrzymane odpowiedzi. Wiersze drugiego typu odpowiadają wywołaniom `PutBack`() i zawierają numery pojemników do odłożenia na półkę. Na końcu zostanie wypisany wiersz o treści "`E`".

Uwaga: w trakcie oceny Twojego rozwiązania z użyciem oficjalnego modułu oceniającego, może ono działać nieco wolniej niż na Twoim komputerze. Różnica ta powinna być jednak nieznaczna. Mimo tego, proponujemy skorzystać z możliwości uruchomienia rozwiązania w systemie sprawdzającym, aby upewnić się, czy Twoje rozwiązanie działa dostatecznie szybko.

Ograniczenia

- Maksymalny czas działania: 7 sekund.
- Dostępna pamięć: 256 MiB.