Karol Pokorski
Treść zadania, Opracowanie

Alan Kutniewski
Program

Dostępna pamięć: 128 MB.

OI, etap III, dzień pierwszy, 28.03.2012

Licytacja

Alojzy i Bajtazar grają w licytację. Do grania w tę grę potrzebny jest bardzo duży zestaw kamyków. Gracze wykonują ruchy na przemian: najpierw Alojzy, potem Bajtazar, znowu Alojzy itd. W danym momencie gry istotne są dwie wartości: aktualna stawka i aktualny rozmiar stosu. Gra zaczyna się od stawki jednego kamyka i pustego stosu. W każdym ruchu gracz wykonuje jeden z następujących ruchów:

- *podwaja stawkę,*
- *potraja stawkę,*
- *pasuje.*

W przypadku, gdy gracz spasuje, cała aktualna stawka wędruje na stos (jest to jedyny sposób powiększenia stosu), a licytacja ponownie zaczyna się od stawki jednego kamyka. Jeżeli gracz spasuje, to następny ruch należy do jego przeciwnika (Alojzy zaczyna tylko całą rozgrywkę). Gracz, który spowoduje przepełnienie stosu (następuje to, gdy na stosie znajduje się n lub więcej kamyków), przegrywa. Jeśli przed ruchem gracza łączna liczba kamyków na stosie i w stawce osiąga lub przekracza n, gracz ten nie może już podwoić ani potroić stawki. Musi spasować, tym samym dokładając stawkę na stos, co powoduje jego przegraną.

Alojzy bardzo często przegrywa. Bajtazar zaproponował mu ciekawe wyzwanie — zamiast samemu grać w licytację, lepiej napisać programy, które będą w nią grały. Niestety, Alojzy nie umie programować. Pomóż mu!

Napisz program, który będzie grał w licytację w imieniu Alojzego przeciw bibliotece napisanej przez Bajtazara.

Ocenianie

We wszystkich przypadkach testowych Twój program będzie mógł wygrać (o ile wykona odpowiednie ruchy) niezależnie od ruchów biblioteki. Twój program otrzyma punkty za dany test tylko wtedy, gdy wygra z biblioteką.

We wszystkich testach zachodzi warunek $1 \leqslant n \leqslant 30\,000$. W 50% przypadków testowych zachodzi dodatkowy warunek $n \leqslant 25$.

Opis użycia biblioteki

Aby użyć biblioteki, należy wpisać na początku programu:

- **C/C++:** `#include "cliclib.h"`
- **Pascal:** `uses pliclib;`

Biblioteka udostępnia następujące funkcje i procedury:

- `inicjuj` — *zwraca liczbę n. Powinna zostać użyta dokładnie raz, na samym początku działania programu.*
  - **C/C++:** `int inicjuj();`
  - **Pascal:** `function inicjuj: longint;`
- `alojzy` — *informuje bibliotekę o ruchu Twojego programu. Jej jedynym parametrem jest liczba całkowita x, która oznacza wykonany ruch: x = 1 oznacza spasowanie, x = 2 — podwojenie stawki, natomiast x = 3 — potrojenie stawki.*
  - **C/C++:** `void alojzy(int x);`
  - **Pascal:** `procedure alojzy(x: longint);`
- `bajtazar` — *funkcja informuje Twój program o ruchu biblioteki. Zwraca jedną liczbę x, która oznacza wykonany ruch. Analogicznie jak w przypadku funkcji* `alojzy`*, x = 1 oznacza pas, x = 2 — podwojenie, natomiast x = 3 — potrojenie stawki.*
  - **C/C++:** `int bajtazar();`
  - **Pascal:** `function bajtazar: longint;`

Po wywołaniu funkcji `inicjuj` *należy naprzemiennie wywoływać funkcje* `alojzy` *oraz* `bajtazar` *(w takiej kolejności). Złamanie protokołu komunikacji zostanie potraktowane jako błędna odpowiedź i spowoduje przyznanie 0 punktów za dany test. W tym zadaniu użycie standardowego wejścia i wyjścia jest* **zabronione**. *Jakakolwiek komunikacja powinna odbywać się tylko za pośrednictwem wyżej podanych funkcji i procedur. Biblioteka zakończy działanie programu automatycznie po zakończeniu gry.*

Rozwiązanie będzie kompilowane wraz z biblioteką przy użyciu następujących poleceń:

- **C:** `gcc -O2 -static cliclib.c lic.c -lm`
- **C++:** `g++ -O2 -static cliclib.c lic.cpp -lm`
- **Pascal:** `ppc386 -O2 -Xs -Xt lic.pas`

Eksperymenty

W katalogu **/home/zawodnik/rozw/lic/** *znajdują się przykładowe pliki bibliotek i przykładowe nieoptymalne rozwiązania ilustrujące sposób ich użycia (można je także pobrać w dziale* **Przydatne zasoby** *w SIO). Program skompilowany z przykładową biblioteką wczytuje ze standardowego wejścia liczbę n, a następnie aż do zakończenia gry wczytuje kolejne ruchy wykonywane przez Bajtazara, symulując zaimplementowaną strategię Alojzego. Na standardowe wyjście diagnostyczne (*`stderr`*) program wypisuje szczegółowy przebieg gry. W testach ocen, zarówno na komputerze zawodnika, jak i w SIO, Bajtazar odpowiada kolejno: pas, podwojenie stawki, potrojenie stawki, pas, podwojenie stawki, potrojenie stawki...*

Ostateczna ocena programów odbędzie się z wykorzystaniem innego zestawu bibliotek.

W przypadku tego zadania w SIO nie jest dostępna opcja **Test programu**.

Przykładowy przebieg programu

C/C++	Pascal	Wynik	Stos	Stawka	Wyjaśnienie
`n = inicjuj();`	`n := inicjuj;`	*15*	*0*	*1*	*Od tego momentu $n = 15$.*
`alojzy(2);`	`alojzy(2);`	—	*0*	*2*	*Twój program podwoił stawkę.*
`bajtazar();`	`bajtazar;`	*2*	*0*	*4*	*Biblioteka odpowiedziała podwojeniem stawki.*
`alojzy(3);`	`alojzy(3);`	—	*0*	*12*	*Twój program potroił stawkę.*
`bajtazar();`	`bajtazar;`	*1*	*12*	*1*	*Biblioteka spasowała. 12 kamyków wędruje na stos, a w stawce ponownie jest tylko 1 kamyk.*
`alojzy(2);`	`alojzy(2);`	—	*12*	*2*	*Twój program podwoił stawkę.*
`bajtazar();`	`bajtazar;`	*3*	*12*	*6*	*Biblioteka potroiła stawkę.*
`alojzy(1);`	`alojzy(1);`	—	*18*	*1*	*Łączna liczba kamyków na stosie i w stawce przekroczyła 15. Twój program jest zmuszony spasować.*

Powyższy przebieg programu jest poprawny, ale nieoptymalny. Twój program nie dostałby punktów za ten test. W szczególności, dla $n = 15$ istnieje możliwość wygranej Alojzego, niezależnie od ruchów Bajtazara.

Rozwiązanie

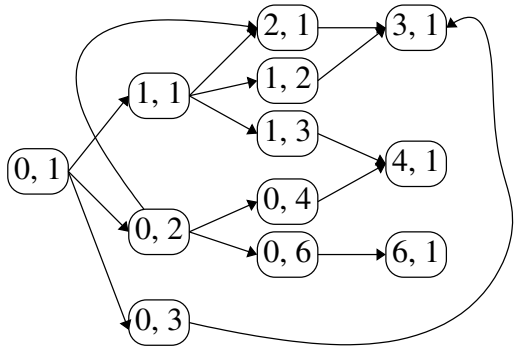
Wprowadzenie

Wyjątkowo problem sformułowany w treści tego zadania nie jest nadmiernie uwikłany w postaci historyjki. Widzimy, że mamy do czynienia z pewną grą. Jakiego typu jest to gra? W tym przypadku najprościej rozumieć grę jako zbiór pozycji i dopuszczalnych ruchów między nimi. Okazuje się, że tego typu relację najwygodniej reprezentować w postaci grafu, w którym wierzchołkami są pozycje, a możliwe ruchy są przedstawiane jako skierowane krawędzie łączące odpowiednie wierzchołki grafu.

Graf pozycji

Pozostaje jeszcze ustalić, czym są pozycje w grze z zadania. Do pełnego opisu aktualnej sytuacji w grze wystarczą nam dwie wartości: rozmiar stosu kamyków oraz stawka. Można zatem oznaczyć wierzchołki grafu pozycji za pomocą uporządkowanych par liczb całkowitych (x, y), gdzie x to aktualna liczba kamieni na stosie, natomiast y to stawka. Przy takich założeniach gra rozpoczyna się na pozycji $(0, 1)$.

Ustalenie dostępnych ruchów dla każdej pozycji jest już bardzo łatwe, należy jednak zachować pewną ostrożność przy oznaczaniu warunków końcowych gry. Ostatecznie, jeśli $x \geqslant n$, to z pozycji (x, y) nie ma żadnych dopuszczalnych ruchów, a w przeciwnym razie z pozycji (x, y) można przejść do pozycji $(x+y, 1)$, a także, o ile $x+y < n$, przejść do pozycji $(x, 2y)$ oraz $(x, 3y)$.



Rys. 1: Graf pozycji dla $n = 3$.

Podział pozycji

Warto jeszcze zauważyć, że gracze są nierozróżnialni. Przy ustalonym stanie gry zarówno Alojzy, jak i Bajtazar mogą wykonać te same ruchy. Pozwala to skupić się na określeniu, dla każdej pozycji, czy jest ona wygrywająca, czy przegrywająca, bez uwzględniania tego, który z graczy w danej chwili wykonuje ruch.

Gracz, który znajduje się na pozycji wygrywającej, może wygrać niezależnie od ruchów przeciwnika (przy założeniu optymalnej gry). Analogicznie, gracz, który znajduje się na pozycji przegrywającej, przegra niezależnie od swoich poczynań, jeśli jego przeciwnik zagra optymalnie.

Powyższa definicja jest poprawna, ale nie przybliża nas do rozwiązania zadania. Spróbujmy scharakteryzować pozycje wygrywające i przegrywające inaczej — za pomocą ruchów, które można z nich wykonać. Na dobry początek warto zauważyć, że wszystkie pozycje, z których nie można wykonać ruchu (pozycje końcowe), są pozycjami wygrywającymi. Istotnie, jeśli gracz znajduje się w pozycji końcowej, to liczba kamieni na stosie jest równa co najmniej n, a zatem jego przeciwnik (który przecież wykonał poprzedni ruch) spowodował przepełnienie stosu, co zgodnie z zasadami gry oznacza jego porażkę. Równoważnie, można by w ogóle nie rozważać pozycji,

w których stos jest przepełniony, i uznać, że gracz, który nie może wykonać ruchu, przegrywa.

Niezależnie od przyjętych założeń co do pozycji końcowych:

- jeśli z danej pozycji istnieje ruch do pozycji przegrywającej, to pozycja ta jest wygrywająca (wystarczy bowiem wykonać ruch do tej pozycji, a wówczas nasz przeciwnik znajdzie się w pozycji przegrywającej),
- jeśli z danej pozycji istnieją tylko ruchy do pozycji wygrywających, to pozycja ta jest przegrywająca (niezależnie od tego, do której pozycji gracz przejdzie, sprowadzi swojego przeciwnika do pozycji wygrywającej).

Powyższe obserwacje są podstawą analizy wielu gier kombinatorycznych. Przykładem może być tutaj zadanie *Kamyki* z XVI Olimpiady Informatycznej [16], w którym analiza pozycji wygrywających i przegrywających również stanowi trzon rozwiązania. Gorąco zachęcamy do lektury opracowania wspomnianego zadania, można bowiem dzięki niemu udoskonalić swoje umiejętności analizy takich gier[1].

Wypracowane prawa rządzące pozycjami wygrywającymi i przegrywającymi są już wystarczające do stwierdzenia, które pozycje są wygrywające, a które przegrywające, a także pozwalają wykonywać ruchy prowadzące do zwycięstwa. Wiadomo przecież, że pozycje końcowe są wygrywające, a obliczenie stanu pozostałych pozycji jest możliwe na podstawie stanów wyznaczonych wcześniej.

Należy tylko opracować kolejność rozpatrywania wierzchołków grafu pozycji, aby mieć pewność, że wszystkie pozycje niezbędne do obliczenia danego stanu zostały już przetworzone. Nie jest to jednak trudne. Otóż wystarczy zauważyć, że graf pozycji jest DAG-iem[2], a szukana kolejność przetwarzania pozycji odpowiada odwróconemu posortowaniu topologicznemu[3] grafu.

Sortowanie topologiczne można zrealizować na co najmniej dwa różne sposoby. Jeden z nich polega na stopniowym usuwaniu z grafu wierzchołków, z których nie wychodzą żadne krawędzie, i dodawaniu ich na początek uporządkowania topologicznego; w drugim zaś wykorzystujemy algorytm przeszukiwania grafu w głąb: za każdym razem, gdy algorytm w pełni przetworzy dany wierzchołek (czyli gdy zostaną odwiedzeni wszyscy jego sąsiedzi i algorytm powraca do rodzica), można dodać ów wierzchołek jako pierwszy do aktualnego uporządkowania topologicznego. Obie te metody łatwo zaimplementować w czasie $O(V + E)$, gdzie V jest liczbą wierzchołków grafu, a E — liczbą jego krawędzi.

Szacowanie liczby możliwych pozycji

Pozostaje przeanalizować, jak duży będzie nasz graf pozycji. Wierzchołków jest w nim tyle, ile par uporządkowanych (x, y), gdzie zarówno x (rozmiar stosu), jak i y (stawka)

[1] Jeszcze więcej o tego typu grach można dowiedzieć się ze strony *Wykładów z Algorytmiki Stosowanej* (`was.zaa.mimuw.edu.pl`) — Wykład 6.

[2] DAG (ang. *directed acyclic graph*) — skierowany graf acykliczny

[3] Posortowanie topologiczne skierowanego grafu acyklicznego jest uporządkowaniem jego wierzchołków, w którym każdy wierzchołek grafu poprzedza wszystkie wierzchołki, do których prowadzą wychodzące z niego krawędzie.

mogą być rzędu n. Daje to $O(n^2)$ stanów, co przy $n \leqslant 30\,000$ jest niewystarczające do uzyskania pełnej punktacji.

Czy to oznacza, że należy zmienić wypracowany pomysł na rozwiązanie? Na szczęście nie. Okazuje się, że uzyskane oszacowanie można poprawić i wówczas stanów będzie istotnie mniej niż rzędu n^2. Należy bowiem zauważyć, że stawka, na początku licytacji równa 1, może się tylko podwajać lub potrajać, a zatem zawsze będzie postaci $2^a \cdot 3^b$, gdzie a i b są liczbami całkowitymi: odpowiednio liczbą podwojeń i potrojeń stawki w aktualnej licytacji. Ponieważ oba te parametry nie przekraczają $\log n$, więc osiągalnych pozycji jest w istocie tylko $O(n \log^2 n)$. Przy przyjętych ograniczeniach na dane wejściowe takie rozwiązanie wystarczało do osiągnięcia wysokich wyników (oscylujących w granicach 80–100 punktów).

Bezpośrednia implementacja powyższego pomysłu znajduje się w pliku `lics3.cpp`.

Rozwiązania alternatywne

Powyższe rozważania okazały się wystarczające do uzyskania poprawnego i dość wydajnego rozwiązania, ale implementacja powyższego pomysłu nie jest ani najprostsza, ani najszybsza. Kluczowym problemem jest wysokie zużycie pamięci — przechowywanie grafu połączeń i posortowania topologicznego jest kosztowne i może przekroczyć nawet stosunkowo duże limity pamięciowe ustalone dla tego zadania.

Przechowywanie całego grafu w pamięci nie jest jednak konieczne. Krawędzie można generować na bieżąco, ponadto zamiast sortowania topologicznego grafu można zastosować algorytm rekurencyjny obliczania stanu pozycji, co w połączeniu ze spamiętywaniem wyników dla wcześniej obliczonych pozycji pozwoli uzyskać rozwiązanie równie efektywne, a przy tym prostsze w implementacji. Zauważmy, że pozycje możemy tu reprezentować za pomocą trójek postaci (x, a, b), gdzie x to rozmiar stosu, a stawka to $2^a \cdot 3^b$, co pozwala wygodnie tablicować wyniki dla pozycji. Programy oparte na tym pomyśle znajdują się w plikach `lic.pas`, `lic1.cpp`, `lic2.pas` oraz `lic3.cpp`.

Część zawodników zapomniała o spamiętywaniu wyników dla wcześniej obliczonych pozycji, co prowadziło do algorytmu wykładniczego. Tego typu rozwiązania zostały zaimplementowane w plikach `lics1.cpp` oraz `lics2.pas`. Na zawodach taki błąd kosztował, zgodnie z uwagą w treści zadania, około połowę możliwych do zdobycia punktów.

Niektórzy finaliści nie poradzili sobie z problemem ograniczenia liczby stanów i pozostali z algorytmem kwadratowym. Takie rozwiązania uzyskiwały na zawodach 75% możliwych do zdobycia punktów.

Rozwiązania polegające na losowaniu ruchów lub heurystycznym ich obliczaniu względem prostych miar zazwyczaj uzyskiwały bardzo mało punktów (najczęściej zero).

Testy

Spośród testów, na jakich były sprawdzane rozwiązania zawodników, połowa to testy poprawnościowe — wybrano w nich takie wartości n, aby gracz rozpoczynający miał

strategię wygrywającą, a bibliotekę ustawiono na grę optymalną (zgodną z programem wzorcowym). W poniższej tabeli takie zachowanie biblioteki zostało oznaczone jako strategia *optymalna*.

W kolejnych testach parametr n stopniowo rośnie, a także stosowane są miejscami inne strategie gry biblioteki: *prawie optymalna*, polegająca na optymalnej grze z małym prawdopodobieństwem wykonania losowego ruchu, oraz *losowa*, w której kilka pierwszych ruchów jest losowych, po czym przełączamy strategię na optymalną.

Nazwa	n	Opis
lic1.in	5	s. optymalna
lic2.in	7	s. optymalna
lic3.in	10	s. optymalna
lic4.in	13	s. optymalna
lic5.in	15	s. optymalna
lic6.in	17	s. optymalna
lic7.in	18	s. optymalna
lic8.in	20	s. optymalna
lic9.in	22	s. optymalna
lic10.in	25	s. optymalna

Nazwa	n	Opis
lic11.in	129	s. prawie optymalna
lic12.in	390	s. optymalna
lic13.in	891	s. prawie optymalna
lic14.in	1 700	s. optymalna
lic15.in	2 000	s. losowa
lic16.in	9 001	s. prawie optymalna
lic17.in	14 000	s. optymalna
lic18.in	21 060	s. losowa
lic19.in	27 400	s. optymalna
lic20.in	29 990	s. optymalna