Piotr Chrząstowski-Wachtel
autor zadania, opracowanie

Adam Borowski
program wzorcowy

Bankomaty

Każdy członek Bajtlandzkiej Kasy Pożyczkowej ma prawo pożyczyć dowolną sumę mniejszą niż 10^{30} bajtlandzkich dukatów, ale musi ją w całości zwrócić do Kasy nie później niż po upływie 7 dni. W sali obsługi klientów Kasy ustawiono 100 bankomatów ponumerowanych od 0 do 99. Każdy bankomat wykonuje tylko jedną operację: wypłaca albo przyjmuje ustaloną kwotę. Bankomat o numerze i wypłaca 2^i dukatów, jeśli i jest parzyste, zaś przyjmuje 2^i dukatów, jeśli i jest nieparzyste. Gdy klient zamierza wypożyczyć ustaloną kwotę, trzeba zbadać, czy będzie mógł ją pobrać, korzystając co najwyżej raz z każdego z bankomatów i jeśli tak, wyznaczyć numery bankomatów, z których należy skorzystać. Trzeba również zbadać, czy będzie mógł ją zwrócić w podobny sposób i jeśli tak, wyznaczyć numery bankomatów, z których należy skorzystać w celu wykonania tej operacji.

Przykład

Klient, który zamierza pożyczyć 7 dukatów, pobiera najpierw 16 dukatów w bankomacie nr 4 i 1 dukata w bankomacie nr 0, a następnie oddaje 8 dukatów w bankomacie nr 3 i 2 dukaty w bankomacie nr 1. Żeby zwrócić pożyczoną kwotę 7 dukatów, pobiera najpierw 1 dukata w bankomacie nr 0, a następnie oddaje 8 dukatów w bankomacie nr 3.

Zadanie

Napisz program, który:

- *wczytuje z pliku tekstowego BAN.IN liczbę klientów n i dla każdego klienta wysokość kwoty, jaką zamierza pożyczyć w Kasie;*
- *dla każdego klienta sprawdza, czy będzie mógł pobrać ustaloną kwotę korzystając co najwyżej raz z każdego bankomatu i jeśli tak, wyznacza numery bankomatów, z których należy skorzystać, oraz czy będzie mógł ją zwrócić w podobny sposób i jeśli tak, wyznacza numery bankomatów, z których należy skorzystać w tym celu;*
- *zapisuje wyniki w pliku tekstowym BAN.OUT.*

Wejście

W pierwszym wierszu pliku wejściowego BAN.IN znajduje się jedna liczba całkowita dodatnia $n \leqslant 1000$. Jest to liczba klientów.

W każdym z kolejnych n wierszy jest jedna liczba całkowita dodatnia, mniejsza niż 10^{30}, zapisana za pomocą co najwyżej 30 cyfr dziesiętnych.

Liczba w i-tym z tych wierszy to wysokość kwoty, którą zamierza pożyczyć klient nr i.

Wyjście

W każdym z 2n kolejnych wierszy pliku wyjściowego BAN.OUT należy zapisać malejący ciąg liczb całkowitych dodatnich z zakresu [0..99] oddzielonych pojedynczymi odstępami albo jedno słowo NIE:

- *w pierwszym wierszu i-tej pary wierszy — numery bankomatów, w porządku malejącym, z których powinien skorzystać klient numer i, by pobrać pożyczkę, albo słowo NIE, gdy nie może jej pobrać zgodnie z ustalonymi regułami;*
- *w drugim wierszu i-tej pary — numery bankomatów, w porządku malejącym, z których powinien skorzystać klient numer i, oddając pożyczkę, albo słowo NIE.*

Przykład

Dla pliku tekstowego BAN.IN:

```
2
7
6338253001141147007483516026 98
```

poprawnym rozwiązaniem jest plik wyjściowy BAN.OUT:

```
4 3 1 0
3 0
NIE
99 3 1
```

Twój program powinien szukać pliku BAN.IN w katalogu bieżącym i tworzyć plik BAN.OUT również w bieżącym katalogu. Plik zawierający napisany przez Ciebie program w postaci źródłowej powinien mieć nazwę BAN.??? gdzie zamiast ??? należy wpisać co najwyżej trzyliterowy skrót nazwy użytego języka programowania. Ten sam program w postaci wykonalnej powinien być zapisany w pliku BAN.EXE.

ROZWIĄZANIE

Zadanie o bankomatach prezentowało ciekawy system zapisywania liczb całkowitych: system o podstawie -2. System ten jest binarnym systemem pozycyjnym, czyli takim, że pozycja cyfry decyduje o związanej z nią wartości, a jedynymi dozwolonymi cyframi są 0 oraz 1, z tym że na $i-$tej pozycji od prawej w zapisie liczby występuje cyfra odpowiadająca wartości $(-2)^i$. Zatem bitom, licząc od prawej, odpowiadają wartości $1, -2, 4, -8, 16, -32, \ldots$. Pojawienie się w zapisie pozycyjnym liczby bitu 1 oznacza dodanie odpowiadającej mu wartości $(-2)^i$, a 0 — pominięcie jej. Kolejne bankomaty odpowiadały kolejnym pozycjom bitów, a wybór bankomatu albo jego pominięcie — jedynce lub zeru na tej pozycji.

Okazuje się, że w tym systemie można zapisać każdą liczbę całkowitą i to dokładnie na jeden sposób. (Rozważamy tylko najkrótszy możliwy zapis.) Kolejnych kilka liczb całkowitych w okolicy zera ma następujące reprezentacje:

-5	1111
-4	1100
-3	1101
-2	10
-1	11
0	0
1	1
2	110
3	111
4	100
5	101

System ten był badany na początku lat 60-tych jako alternatywny do układu dwójkowego w procesorach komputerowych. Miał on kilka zalet, np. ujednolicenie zapisu i działań na liczbach dodatnich i ujemnych, brak wyróżnionych bitów, a także kilka wad, np. niesymetryczność zakresu liczb dodatnich i ujemnych w n-bitowej reprezentacji. Wadą tego systemu był też sposób obliczania liczby przeciwnej do danej, co okazało się niebanalną operacją. Widać, że postać każdej liczby w zapisie o podstawie -2 oraz postać liczby przeciwnej do niej różnią się od siebie zasadniczo.

Zadanie o bankomatach polegało właśnie na znalezieniu reprezentacji bitowej zadanej liczby i liczby przeciwnej do niej w układzie o podstawie -2.

Algorytm przedstawienia dowolnej nieujemnej liczby x w układzie pozycyjnym o podstawie $b \geqslant 2$ jest następujący. W tablicy cyfr $C[0..\mathit{liczbacyfr}-1]$ **of** $0..b-1$ umieszczamy od końca kolejne cyfry przedstawienia:

```
1: for i:=0 to liczbacyfr − 1 do
2:   begin
3:   C[i] := x mod b;
4:     x := x div b
5: end
```

Dzielimy zatem liczbę x przez b tyle razy, ile cyfr przedstawienia sobie zażyczymy. W tablicy cyfr C od prawej do lewej odnotowujemy reszty z tego dzielenia. Jeżeli wartość x po kilku iteracjach spadnie do zera, to pozostała część tablicy, aż do lewego końca, zostanie uzupełniona zerami. Możemy też wyjść wcześniej z pętli, zmieniając jej nagłówek na **while** $x \neq 0$ **do**. W tej wersji trzeba albo wypełnić na początku tablicę C zerami, albo pamiętać, gdzie się zaczyna zapis liczby.

Powyższy algorytm dla $b = 2$ nazywany jest niekiedy algorytmem *zamiany liczby z układu dziesiętnego na dwójkowy*, co jest mylną nazwą, bo z układem dziesiętnym ma on niewiele wspólnego. Może tyle tylko, że do reprezentowania liczby jesteśmy przyzwyczajeni używać układu dziesiętnego. W algorytmie bowiem odwołujemy się do tego, co z liczby pozostanie w kolejnych przebiegach pętli algorytmu, a nie do rozwinięcia dziesiętnego tej liczby i jej pozostałości „wewnątrz" algorytmu.

Ponieważ mamy znaleźć przedstawienie liczb w układzie o podstawie -2, więc najprościej byłoby użyć tego algorytmu dla $b = -2$. Problem polega na tym, że nie jest jasne, jakie znaczenie nadać wyrażeniom x **mod** b oraz x **div** b, gdy b lub x są ujemne. Aby wyjaśnić tę kwestię można spróbować użyć dwóch metod: napisać prosty program w Pascalu i najzwyczajniej w świecie przyjrzeć się kilku wartościom,

co oczywiście nie da stuprocentowej pewności co do poprawności, ale przynajmniej pozwoli się upewnić co do koncepcji. Drugi sposób to oczywiście spróbować udowodnić poprawność algorytmu dla ujemnych b.

Napisanie w Pascalu lub w C programu, który realizuje pierwszy pomysł dla $b = -2$ przynosi rozczarowanie: w tablicy C, np. dla $x = 11$, pojawiają się 3 wartości: $-1, 0, 1$ — zupełnie nie to, czego się spodziewaliśmy.

Spróbujmy zatem drugiej metody — dowodu poprawności algorytmu. Oczywiście możemy to potraktować w formie żartu, bo skoro pierwsza metoda zawiodła, to po co dalej rozmyślać? Nie jest to jednak do końca taki czczy żart, bo jeśli przyjrzymy się wynikom, to zobaczymy, że te rozwinięcia, które nie zawierają minus jedynek są dobre, a te, które zawierają — też są sensowne, tyle że sugerują wzięcie odpowiednich wartości z minusem. Na przykład liczba 11 ma rozwinięcie (-1)0(-1)1, czyli jakby $(-1)*(-8)+0*(4)+(-1)*(-2)+1*(1) = 11$. Coś się tu jednak zgadza. W innych przypadkach jest podobnie. Może zatem wystarczy się jakoś uwolnić od tych minus jedynek i w ogóle będzie dobrze? Wróćmy do pomysłu dowodu algorytmu zamiany podstawy układu pozycyjnego i zobaczmy, gdzie się nam dowód załamie.

Jaka jest interpretacja operacji **div** i **mod**? Oczywiście chodzi o iloraz i resztę dzielenia całkowitoliczbowego, czyli o takie dwie liczby d i r, że

$$x = db + r, \qquad \text{gdzie} \quad 0 \leqslant r < |b|. \tag{1}$$

Liczby d i r są wyznaczone jednoznacznie przez dowolne x oraz b, takie że $|b| > 1$.

Twierdzenie 1: Przy przyjętej interpretacji operacji **div** i **mod** algortym z poprzedniej strony jest poprawny dla każdej podstawy b takiej, że $|b| \geqslant 2$.

Dowód: Indukcja ze względu na $|x|$. Zacznijmy od przypadku $b \geqslant 2$ oraz $x \geqslant 0$. Najpierw odnotujmy, że liczba 0 daje właściwe przedstawienie przy każdej podstawie $b > 1$: wszystkie operacje **div** i **mod** dają w wyniku zero, więc tablica C zostanie wypełniona samymi zerami. Teraz załóżmy, że nasz algorytm działa poprawnie dla wszystkich wartości nieujemnych mniejszych niż pewne $x > 0$ (czyli wypełnia dla takich wartości tablicę C właściwymi cyframi). Udowodnimy, że w tym przypadku będzie poprawnie działał również dla liczby x.

Prześledźmy, co się stanie, gdy wykonamy pierwszy obrót pętli naszego algorytmu. Do tablicy C na pozycji 0 zostanie wpisany wynik x **mod** b, a liczba x stanie się równa x **div** b. Ponieważ $b > 1$, więc liczba x **div** b jest mniejsza niż x. Zatem na mocy założenia indukcyjnego liczba x **div** b zostanie w dalszej części algorytmu prawidłowo rozwinięta w komórkach $C[1], C[2], \ldots$, czyli do tablicy zostaną wpisane właściwe cyfry rozwinięcia x**div**b przy podstawie b, tyle że przesunięte o 1 w lewo. Ze względu na to, że przesunięcie o 1 w lewo oznacza w każdym potęgowym układzie pozycyjnym przemnożenie przez b, otrzymana liczba będzie równa

$$b \cdot (x \textbf{ div } b) + (x \textbf{ mod } b) = x,$$

co było do pokazania (na razie dla $x \geqslant 0$, $b \geqslant 2$).

Teraz najważniejsze odkrycie pozwalające nam przenieść ten dowód na ujemne wartości b oraz x. Zauważmy, że jedynym miejscem dowodu, w którym zakładaliśmy, że $b \geqslant 2$ i $x \geqslant 0$ było wykorzystanie kroku indukcyjnego: chodziło o to, żeby liczba

x **div** b była mniejsza od x. Reszta rachunków nie wymagała ani dodatności b, ani nieujemności x.

Jak zatem zabrać się za ujemne wartości b i x? Wystarczy wymusić istotny postęp w „zmniejszaniu" się liczby x. W tym celu zauważmy, że dla $b \leqslant -2$, dla każdego x, poza jednym przypadkiem, wartość $|x$ **div** $b|$ jest mniejsza, niż $|x|$. Tym jedynym przypadkiem jest $x = -1$. Wtedy jednak x **div** $b = 1$ i w zasadzie można na tym poprzestać, bo w kolejnym kroku 1 **div** $b = 0$. Czyli poza -1, zarówno dla dodatnich, jak i dla ujemnych liczb x, mamy $|x$ **div** $b| < |x|$. Ze względu na to, że również dla -1 w następnym kroku osiągniemy zero uzyskując właściwą reprezentację: $-1 = (0 \cdots 01(b-1))_b$, rozumowanie możemy tu zakończyć. Formalnie powiększamy bazę indukcji o przypadki $x = -1$ i $x = 1$. Następnie zakładamy, że teza zachodzi dla wszystkich liczb co do modułu mniejszych od $|x|$, po czym przeprowadzamy wnioskowanie identyczne z poprzednim, które przekona nas, że algorytm działa poprawnie dla każdej liczby całkowitej x i dla każdej liczby b o module większym od 1. □

Zaraz, zaraz! To dlaczego nasz program w Pascalu dał złe wyniki? Skąd się wzięły te minus jedynki? Ja tego szczerze powiedziawszy nie rozumiem. To znaczy nie rozumiem, dlaczego tak wybitny informatyk, jakim jest Niklaus Wirth, projektując Pascala, w tak dziwaczny sposób zdefiniował operacje **div** i **mod**. Okazuje się, że dla liczb ujemnych pascalowe odpowiedniki wyników operacji **mod** i **div** nie mają nic wspólnego ze wzorem (1). Na przykład, zgodnie ze wzorem (1), (-7) **div** $(-2) = -4$, (-7) **mod** $(-2) = 1$, podczas gdy w Pascalu odpowiednie wyniki, to 3 i -1.

Widać więc, że przy nieograniczonej liczbie bankomatów każdą sumę dukatów da się pożyczyć i oddać, a algorytm wyboru bankomatów jest bardzo prosty. Jego istotę (z pominięciem problemów związanych z realizacją omawianych operacji) oddaje następujący kod poprawny dla każdej podstawy b:

```
1: for i:=0 to LiczbaBankomatow - 1 do
2:   begin
3:     Bankomat[i] := x mod (abs(b)); {u nas b = -2 }
4:     x := (x - Bankomat[i]) div b {odejmujemy, żeby uniknąć niejasności
                                                  z operacją div}
5:   end
```

Rzecz jasna, gdyby po zakończeniu pętli wartość x była różna od zera, to oznaczałoby to, że nie starcza nam bankomatów na wyrażenie aż tak dużej co do modułu liczby i byłaby to jedyna przyczyna, dla której odpowiedzią powinno być NIE. W rzeczywistości ograniczenia w zadaniu były tak dobrane, żeby na końcach przedziału występowały wartości niewyrażalne na 100 pozycjach w układzie o podstawie (-2).

Tym razem obowiązek zrealizowania operacji **div** i **mod** spoczywał na zawodnikach, bowiem zakres liczb przekraczał rozmiar wszelkich standardowych typów, w których te operacje są zdefiniowane na poziomie języka (zresztą, o czym była mowa, niezbyt szczęśliwie z punktu widzenia naszych potrzeb). Nie było to trudne, bo pierwsza operacja wymaga sprawdzenia jednego bitu, a druga — generując na przemian wyniki dodatnie i ujemne — sprowadzała się do ucinania ostatniej cyfry z ewentualnym odjęciem jedynki od wyniku dla argumentu ujemnego.

TESTY

Testy były podzielone na 11 grup.

- BAN0.IN — test z treści zadania.
- BAN1.IN — jedna liczba równa 1.
- BAN2.IN — 3 niewielkie liczby jednobajtowe.
- BAN3.IN — 100 liczb mieszczących się w typie **longint** (4 bajty).
- BAN4.IN — wszystkie liczby od 1 do 1000.
- BAN5.IN — różne liczby, w tym kilka trudnych, na granicy reprezentowalności.
- BAN6.IN — 442 liczby postaci $2^i, 2^i + 1, 2^i - 1, F(i)$, gdzie $F(i)$, to i-ta liczba Fibonacciego.
- BAN7.IN — 978 losowych liczb o liczbie cyfr z przedziału 1..30.
- BAN8.IN — 1000 losowych liczb o maksymalnej długości 30 cyfr.
- BAN9.IN — 1000 największych liczb trzydziestocyfrowych cyfr (harmonia dziewiątek z różnymi końcówkami).
- BAN10.IN — 1000 losowych liczb 28–30-cyfrowych