Jakub Radoszewski — Treść zadania, Opracowanie
Łukasz Bieniasz-Krzywiec — Program

Dostępna pamięć: 128 MB. *OI, Etap III, dzień pierwszy, 14.04.2010*

Żabka

Wzdłuż bajtockiego strumyczka stoi n kamieni. Są one położone kolejno w odległościach $p_1 < p_2 < \ldots < p_n$ od źródła strumyczka. Na jednym z tych kamieni stoi żabka, która właśnie ćwiczy się w skakaniu. W każdym skoku żabka skacze na k-ty najbliższy kamień (od kamienia, na którym siedzi). Dokładniej, jeżeli żabka w danej chwili siedzi na kamieniu w położeniu p_i, to po wykonaniu skoku będzie siedziała na takim kamieniu p_j, że:

$$\left|\left\{p_a \ : \ |p_a - p_i| < |p_j - p_i|\right\}\right| \leqslant k \qquad \text{oraz} \qquad \left|\left\{p_a \ : \ |p_a - p_i| \leqslant |p_j - p_i|\right\}\right| > k.$$

Pierwsza z powyższych nierówności oznacza, że liczba punktów $p_a \in \{p_1, p_2, \ldots, p_n\}$, których odległość od punktu p_i jest mniejsza niż odległość między punktami p_i a p_j, jest nie większa niż k. Podobnie, druga z nierówności oznacza, że liczba punktów $p_a \in \{p_1, p_2, \ldots, p_n\}$, których odległość od punktu p_i jest nie większa niż odległość między punktami p_i a p_j, jest większa niż k.

W przypadku, gdy istnieje więcej niż jedno takie p_j, żabka wybiera skrajnie lewą spośród takich pozycji. Na którym kamieniu żabka znajdzie się po wykonaniu m skoków, w zależności od tego, z którego kamienia zaczyna?

Wejście

Pierwszy wiersz standardowego wejścia zawiera trzy liczby całkowite n, k oraz m ($1 \leqslant k < n \leqslant 1\,000\,000$, $1 \leqslant m \leqslant 10^{18}$), pooddzielane pojedynczymi odstępami i reprezentujące odpowiednio: liczbę kamieni, parametr skoku żabki oraz liczbę zaplanowanych skoków żabki. Drugi wiersz zawiera n liczb całkowitych p_j ($1 \leqslant p_1 < p_2 < \ldots < p_n \leqslant 10^{18}$), pooddzielanych pojedynczymi odstępami i oznaczających położenia kolejnych kamieni wzdłuż strumyczka.

Wyjście

Twój program powinien wypisać na standardowe wyjście dokładnie jeden wiersz zawierający n liczb całkowitych $r_1, r_2, \ldots, r_n$ z przedziału $[1, n]$, pooddzielanych pojedynczymi odstępami. Liczba r_i oznacza numer kamienia, na którym żabka zakończy skakanie, jeśli rozpocznie je z kamienia o numerze i (zgodnie z kolejnością z wejścia).

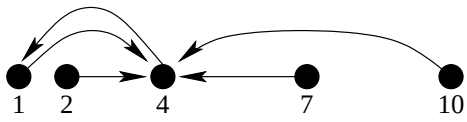
Przykład

Dla danych wejściowych:

```
5 2 4
1 2 4 7 10
```

poprawnym wynikiem jest:

```
1 1 3 1 1
```



Na rysunku pokazano, jak żabka skacze z poszczególnych kamieni (w jednym skoku).

Rozwiązanie

Wprowadzenie

Rozwiązanie niniejszego zadania w naturalny sposób dzieli się na dwie części, z których pierwsza jest związana z parametrem k, a druga z parametrem m. W Fazie I dla każdego z kamieni $p_1, p_2, \ldots, p_n$ znajdujemy k-ty najbliższy mu kamień, co reprezentujemy za pomocą tablicy f:

$$f[i] = j \quad \Leftrightarrow \quad p_j \text{ to } k\text{-ty najbliższy kamień od kamienia } p_i. \tag{1}$$

Tablica f pokazuje zatem, jak żabka skacze z poszczególnych kamieni (w jednym skoku). W Fazie II dla każdego kamienia p_i sprawdzamy, gdzie żabka znajdzie się po wykonaniu m skoków, począwszy od tego kamienia, czyli obliczamy wartość:

$$f^m[i] \;=\; \underbrace{f[f[\ldots f[}_{m \text{ razy}} i] \ldots]]. \tag{2}$$

Widać wyraźnie, że podane fazy możemy rozpatrywać niezależnie. Dla każdej z nich omówimy sposób dojścia od najprostszego (a zarazem najmniej efektywnego) rozwiązania do algorytmu wchodzącego w skład rozwiązania wzorcowego.

W dalszym opisie kamień w położeniu p_i będziemy często nazywali po prostu i-tym kamieniem.

Faza I

Metoda 1a: złożoność czasowa $O(n^2 \log n)$

Aby obliczyć $f[i]$, czyli numer k-tego najbliższego kamienia od i-tego kamienia, możemy na przykład posortować wszystkie kamienie względem odległości od i-tego kamienia, po czym jako $f[i]$ przyjąć numer k-tego kamienia w tym porządku. W ogólności ten pomysł brzmi całkiem rozsądnie, ale musimy poświęcić chwilę na analizę dosyć technicznej definicji *k-tego najbliższego kamienia* z treści zadania, zwracając dodatkowo szczególną uwagę na remisy.

Spróbujmy zacząć od czysto intuicyjnego podejścia, ignorując skomplikowanie wyglądające nierówności z treści zadania. Niech t będzie tablicą złożoną z par postaci:

$$t = ((|p_j - p_i|,\ j) \ :\ j = 1, 2, \ldots, n),$$

reprezentującą odległości wszystkich kamieni od i-tego kamienia. Załóżmy, że t jest uporządkowana rosnąco, przy czym pary są porównywane leksykograficznie po współrzędnych (tzn. porównywane w pierwszej kolejności po pierwszej współrzędnej, a w razie remisu — po drugiej). Zauważmy, że w przypadku równych pierwszych współrzędnych para reprezentująca kamień położony bardziej na lewo znajduje się w tablicy wcześniej. Czy wówczas k-ty element posortowanej tablicy będzie wskazywał na kamień reprezentujący $f[i]$?

Odpowiedź na postawione pytanie jest negatywna, ale zaproponowane rozwiązanie nie jest bardzo odległe od poprawnego — musimy tylko dopracować kilka szczegółów.

Przede wszystkim wykluczmy z rozważań sam i-ty kamień: możemy go usunąć z tablicy t, a także wyeliminować z nierówności z zadania, otrzymując:

$$\begin{aligned} |\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| < |p_j - p_i|\}| &< k \\ |\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| \leqslant |p_j - p_i|\}| &\geqslant k. \end{aligned}$$

Kolejne spostrzeżenie jest takie, że w ustalonej odległości $d > 0$ od i-tego kamienia mogą znajdować się co najwyżej dwa inne kamienie: jeden po lewej, a drugi po prawej jego stronie. Zauważmy, że tylko skrajnie lewy spośród tych dwóch kamieni może wyznaczać wartość $f[i]$. W takim razie, wystąpienie skrajnie prawego w t możemy zastąpić wystąpieniem skrajnie lewego.

Niech teraz $t[k] = (|p_j - p_i|, \ j)$ będzie wystąpieniem j-tego kamienia w zmodyfikowanej w opisany sposób tablicy t. Wówczas

$$|\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| < |p_j - p_i|\}| \in \{k-1, k-2\},$$

natomiast

$$|\{p_a \ : \ a \ \neq i \ \wedge \ |p_a - p_i| \leqslant |p_j - p_i|\}| \in \{k, k+1\},$$

przy czym dokładna wartość w każdym przypadku zależy od tego, czy jest to pierwsze (lub jedyne), czy drugie wystąpienie j-tego kamienia. To pokazuje, że $f[i] = j$.

Poniższy pseudokod ilustruje opisane podejście. Wykorzystujemy w nim procedurę sortującą tablicę t w czasie $O(n \log n)$, patrz np. książka [21]. Dostęp do poszczególnych elementów par w tablicy uzyskujemy za pomocą pól *first* oraz *second*. Zakładamy wreszcie możliwość umieszczania nowych elementów na końcu tablicy t (operacja *insert*), co możemy zaimplementować, używając odpowiednio dużej, statycznej tablicy oraz zmiennej reprezentującej liczbę zajętych początkowych pól. Dodajmy dla jasności, że t jest indeksowana od jedynki.

```
1:  Algorytm 1a
2:    for i := 1 to n do begin
3:      t := ∅;
4:      for j := 1 to n do
5:        if j ≠ i then t.insert((|p_j − p_i|, j));
6:      sort(t);
7:      for j := 2 to n − 1 do
8:        if t[j].first = t[j − 1].first then t[j] := t[j − 1];
9:      f[i] := t[k].second;
10:   end
11:   return f;
```

Złożoność czasowa Algorytmu 1a to $O(n^2 \log n)$.

Metoda 1b: złożoność czasowa $O(n \cdot k)$

Stosunkowo proste usprawnienie Metody 1a możemy uzyskać, jeśli zauważymy, że *im dalej od i-tego kamienia, tym odległości od niego są większe*. To oczywiste stwierdzenie

implikuje, że zawartość tablicy t otrzymujemy w wyniku scalenia dwóch następujących list, uporządkowanych względem odległości od kamienia p_i:

$$L_1 = ((p_i - p_{i-1}, i-1), \ldots, (p_i - p_1, 1)) \quad \text{oraz} \quad L_2 = ((p_{i+1} - p_i, i+1), \ldots, (p_n - p_i, n)).$$

Podczas scalania musimy pamiętać o tym, żeby w przypadku remisu do tablicy t wstawić dwie kopie skrajnie lewego spośród kamieni równoodległych od i-tego kamienia. Jeśli zauważymy, że list L_1 i L_2 nie musimy konstruować explicite (wystarczą nam dwa wskaźniki przemieszczające się po tablicy p wszystkich kamieni) oraz że scalanie możemy przerwać w momencie, gdy wynikowa lista ma już co najmniej k elementów, to otrzymamy algorytm o złożoności czasowej $O(n \cdot k)$.

```
1:  Algorytm 1b
2:    for i := 1 to n do begin
3:      t := ∅;
4:      a := i − 1;  b := i + 1;
5:      while (a ⩾ 1) and (b ⩽ n) and (size(t) < k) do begin
6:        if p_i − p_a < p_b − p_i then begin
7:          { Kamień a jest bliższy. }
8:          t.insert((p_i − p_a, a));  a := a − 1;
9:        end else if p_i − p_a > p_b − p_i then begin
10:         { Kamień b jest bliższy. }
11:         t.insert((p_b − p_i, b));  b := b + 1;
12:       end else begin
13:         { Mamy remis, więc dwukrotnie wstawiamy kamień a. }
14:         t.insert((p_i − p_a, a));  t.insert((p_i − p_a, a));
15:         a := a − 1;  b := b + 1;
16:       end
17:     end
18:     { Po jednej stronie zużyliśmy już wszystkie kamienie, }
19:     { po drugiej jeszcze nie. }
20:     while (a ⩾ 1) and (size(t) < k) do begin
21:       t.insert((p_i − p_a, a));  a := a − 1;
22:     end
23:     while (b ⩽ n) and (size(t) < k) do begin
24:       t.insert((p_b − p_i, b));  b := b + 1;
25:     end
26:     f[i] := t[k].second;
27:   end
28:   return f;
```

Metoda 1c: złożoność czasowa $O(n \log n \log p_n)$

Metoda 1b jest efektywna, jeśli żabka postanawia wykonywać jedynie niezbyt długie skoki, tzn. wartość parametru k jest niewielka. Teraz opiszemy metodę, która spisuje się całkiem dobrze także, gdy k jest duże. Będzie ona oparta na wyszukiwaniu binarnym, i to niejednym.

Możemy mianowicie rozpocząć wyznaczanie $j = f[i]$ od znalezienia wartości $d = |p_j - p_i|$. Innymi słowy, poszukujemy takiej granicznej wartości d, że w odległości nie większej niż d od i-tego kamienia znajduje się co najmniej k innych kamieni, ale w odległości nie większej niż $d-1$ od i-tego kamienia jest już mniej niż k kamieni. Wartość d wyszukamy binarnie w przedziale $[1, p_n]$.

Pozostaje pytanie, jak wyznaczyć liczbę kamieni oddalonych od i-tego kamienia nie więcej niż o d. W tym celu ponownie stosujemy wyszukiwanie binarne (a nawet dwa), aby znaleźć numery skrajnych kamieni położonych po lewej i po prawej stronie i-tego kamienia, których odległości od tego kamienia nie przekraczają d.

Poniżej implementacja tej metody wyznaczania wartości d o złożoności czasowej $O(\log n \log p_n)$.

```
1:  Algorytm 1c, wyznaczanie granicznej odległości d
2:    function na_lewo(i, odl)
3:    begin
4:      lo' := 1; hi' := i - 1;
5:      while lo' < hi' do begin
6:        s' := (lo' + hi') div 2;
7:        if p_i - p_s' ⩽ odl then hi' := s'
8:        else lo' := s' + 1;
9:      end
10:     return lo';
11:   end
12:
13:   function na_prawo(i, odl)
14:   { Analogicznie do na_lewo. }
15:
16:   function oblicz_d(i)
17:   begin
18:     lo := 1; hi := p_n
19:     while lo < hi do begin
20:       s := (lo + hi) div 2;
21:       if na_prawo(i, s) - na_lewo(i, s) ⩾ k then hi := s
22:       else lo := s + 1;
23:     end
24:     return lo;
25:   end
```

Zauważmy na koniec, że znając wartość d, możemy jeszcze raz wywołać funkcje *na_lewo* i *na_prawo*, tym razem już z właściwym argumentem $odl = d$, i ze zwróconych przez nie wyników wybrać ten, który reprezentuje kamień położony dalej od i-tego (lub skrajnie lewy w przypadku remisu). Poprawność tego stwierdzenia wynika stąd, że jeśli odległości skrajnych kamieni od i-tego są różne, to $\text{na\_prawo}(i, d) - \text{na\_lewo}(i, d) = k$, a w przeciwnym przypadku ta różnica jest równa k lub $k+1$. Stosowne dokończenie implementacji poniżej.

```
1:  Algorytm 1c, dokończenie
2:    for i := 1 to n do
```

```
3:  begin
4:    d := oblicz_d(i);
5:    a := na_lewo(i, d);  b := na_prawo(i, d);
6:    if p_i − p_a ⩾ p_b − p_j then f[i] := a
7:    else f[i] := b;
8:  end
9:  return f;
```

Złożoność czasowa Metody 1c to $O(n \log n \log p_n)$. Jest to już całkiem niezły wynik jak na ograniczenia z zadania: $n \leqslant 10^6$, $p_n \leqslant 10^{18}$. Niemniej jednak, także i ten rezultat jeszcze da się poprawić.

Metoda 1d: złożoność czasowa $O(n)$

W dotychczasowych rozwiązaniach wyznaczaliśmy jedynie k-te najbliższe kamienie; tym razem pójdziemy o krok dalej i przyjrzymy się temu, jak wygląda *zbiór* k kamieni najbliższych i-temu kamieniowi (remisy rozstrzygamy zgodnie z regułą skrajnie lewego). Już w Metodzie 1c zauważyliśmy, że jest to przedział, złożony z pewnej liczby kamieni położonych na lewo od i-tego oraz pewnej liczby kamieni leżących na prawo od i-tego. Oznaczmy ten przedział, wraz z samym i-tym kamieniem, przez $I[i] = [a\,..\,a+k]$.

Aby efektywnie wyznaczać przedziały $I[i]$, zastanowimy się, jaka jest zależność pomiędzy $I[i-1]$ a $I[i]$. Zauważmy, że jest to także pewne novum w stosunku do dotychczasowych podejść, w których każdy kamień rozpatrywaliśmy osobno. Okazuje się, że zachodzi następujący, całkiem intuicyjny fakt.

Fakt 1. *Jeśli* $I[i-1] = [a\,..\,a+k]$ *oraz* $I[i] = [b\,..\,b+k]$, *to* $a \leqslant b$.

Dowód: Załóżmy nie wprost, że $b < a$. Wówczas mamy $(b+k+1) \in I[i-1]$, ale $(b+k+1) \notin I[i]$. Korzystając z tego spostrzeżenia, otrzymujemy:

$$|p_{b+k+1} - p_i| \overset{(1)}{<} |p_{b+k+1} - p_{i-1}| \overset{(2)}{<} |p_{a-1} - p_{i-1}| \overset{(3)}{<} |p_{a-1} - p_i| \overset{(4)}{\leqslant} |p_b - p_i|.$$

Powyższe nierówności uzasadniamy następująco:

(1) Ponieważ $i - 1 < i \ < b + k + 1$.

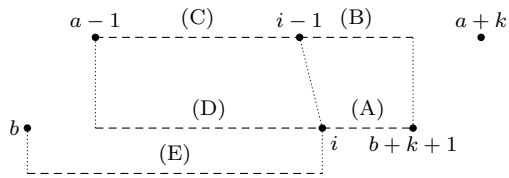
(2) Ponieważ $(b + k + 1) \in I[i-1]$, ale $(a-1) \notin I[i-1]$.

(3) Ponieważ $a - 1 < i - 1 < i$.

(4) Ponieważ $b < a$.

Graficzne przedstawienie tego ciągu nierówności, przy oznaczeniu kolejnych z powyższych wartości bezwzględnych różnic przez (A), (B), (C), (D), (E), przedstawia rys. 1.

Aby zakończyć dowód, wystarczy zauważyć, że uzyskany ciąg nierówności stanowi sprzeczność z faktem, iż $b \in I[i]$, ale $(b + k + 1) \notin I[i]$. ■



Rys. 1: Ilustracja ciągu nierówności: $(A) \overset{(1)}{<} (B) \overset{(2)}{<} (C) \overset{(3)}{<} (D) \overset{(4)}{\leqslant} (E)$.

Musimy jeszcze odpowiedzieć na dwa pytania.

1. *Jak wykorzystać Fakt 1 do efektywnego wyznaczania przedziałów $I[i]$?* W tym celu każdorazowo zaczynamy od $I[i] = I[i-1]$, po czym przesuwamy ten przedział o jeden w prawo, dopóki jest to wskazane. Łatwo widać, że przedział $[a..a+k]$ należy zamienić na $[a+1..a+k+1]$, jeżeli $(a+k+1)$-szy kamień jest położony bliżej i-tego niż znajdujący się aktualnie w przedziale kamień a-ty, czyli:
$$p_{a+k+1} - p_i < p_i - p_a.$$

2. *Po co nam przedziały $I[i]$, czyli jak za ich pomocą obliczać wartości $f[i]$?* Czytelnik zaznajomiony z poprzednimi metodami szybko dostrzeże, że $f[i]$ odpowiada jednemu z końców przedziału $I[i]$. Jeżeli odległości kamieni odpowiadających końcom tego przedziału od kamienia i-tego są różne, to $f[i]$ uzyskujemy, biorąc dalszy z nich. W przeciwnym przypadku, ze względu na kryterium rozstrzygania remisów z treści zadania, jako wynik przyjmujemy kamień skrajnie lewy. Uzasadnienie poprawności tego spostrzeżenia pozostawiamy Czytelnikowi (wbrew pozorom, na ewentualne remisy należy zwrócić uwagę w obydwu przypadkach).

Algorytm 1d otrzymujemy jako bezpośrednią konsekwencję odpowiedzi na powyższe pytania, patrz implementacja poniżej. Na pierwszy rzut oka niepokoić może jego złożoność czasowa: mamy dwie zagnieżdżone pętle, z których każda wykonuje co najwyżej n obrotów. Nietrudno jednak zauważyć, że łączna liczba obrotów tych pętli będzie liniowa względem n, jako że każdy obrót pętli **while** powoduje zwiększenie wartości zmiennej a o jeden, a zmienna ta przyjmuje wartości z przedziału $[1, n-k]$. To rozumowanie, będące bardzo prostym przypadkiem *analizy kosztu zamortyzowanego* (patrz np. książka [21]), pokazuje, że złożoność czasowa poniższego algorytmu to $O(n)$. Dodajmy dla ścisłości, że złożoność pamięciowa tego algorytmu, a także wszystkich wcześniej zaprezentowanych, to $O(n)$.

```
1: Algorytm 1d
2:   a := 1;
3:   f[1] := k + 1;
4:   for i := 2 to n do begin
5:     while (a + k + 1 ⩽ n) and (p_{a+k+1} − p_i < p_i − p_a) do
6:       a := a + 1;
7:     if p_i − p_a ⩾ p_{a+k} − p_i then f[i] := a
8:     else f[i] := a + k;
9:   end
```

```
10:  return f;
```

Porównując Algorytm 1d z mniej efektywnymi metodami, otrzymujemy kolejne potwierdzenie częstego zjawiska, że najefektywniejszy algorytm dla danego problemu cechuje się jednocześnie najprostszą implementacją. Co bynajmniej nie oznacza, że jest najprostszy do wymyślenia!

Faza II

W poprzednich sekcjach przedstawiliśmy kilka różnych algorytmów pozwalających obliczyć $f[i]$ dla $i = 1, 2, \ldots, n$, która to funkcja reprezentuje miejsca docelowe pojedynczych skoków żabki z poszczególnych kamieni. Pod koniec udało nam się już uzyskać całkiem efektywne rozwiązania. Przyszła pora, aby obliczyć miejsca docelowe po seriach skoków długości m, czyli wartości $r[i] := f^m[i]$ dla $i = 1, 2, \ldots, n$.

Metoda 2a: złożoność czasowa $O(n \cdot m)$

Zacznijmy od najprostszego możliwego algorytmu: wartość $f^m[i]$ możemy obliczyć wprost z definicji (2). Złożoność czasowa takiego podejścia to ewidentnie $\Theta(n \cdot m)$ — rzut oka na ograniczenia z zadania pokazuje, że jest to metoda dalece niewystarczająca.

Metoda 2b: złożoność czasowa $O(n^2)$

Przyjrzyjmy się sekwencji kolejnych wartości postaci:

$$i,\ f[i],\ f[f[i]],\ f[f[f[i]]], \ldots$$

Czy można w tym ciągu wychwycić jakąś regularność, która mogłaby pozwolić na szybsze obliczenie $f^m[i]$?

Otóż tak, mianowicie z zasady szufladkowej Dirichleta wynika, że wśród pierwszych $n+1$ wartości $f^j[i]$ ($j = 1, 2, \ldots$) jakaś liczba musi się powtórzyć, gdyż wszystkie te liczby są z zakresu $1..n$. Niech p i q wskazują na pierwsze takie powtórzenie (czyli q najmniejsze możliwe). Wówczas:

$$f^p[i] = f^q[i], \quad f^{p+1}[i] = f^{q+1}[i], \quad f^{p+2}[i] = f^{q+2}[i], \ldots$$

czyli od q-tego elementu fragment $f^p[i], f^{p+1}[i], \ldots, f^{q-1}[i]$ będzie powtarzał się w nieskończoność. To daje prostą receptę na wyznaczenie $f^m[i]$. Znajdujemy wartości p oraz q; jeżeli $m < q$ to przy okazji obliczyliśmy także $f^m[i]$, a w przeciwnym razie możemy odczytać żądaną wartość na podstawie $(m-q) \bmod (q-p)$, czyli traktując m modulo długość cyklu.

Poniższy pseudokod przedstawia implementację tego podejścia w złożoności czasowej $O(n^2)$. Podobnie jak w metodach z Fazy I, wykorzystujemy w nim tablicę indeksowaną od jedynki, do której możemy dokładać elementy na koniec (zmienna t). Łącząc pomysły z Metod 2a oraz 2b, można dokonać kosmetycznej poprawki złożoności czasowej do $O(n \cdot \min(n, m))$, co pozostawiamy Czytelnikowi do przemyślenia.

```
1:  Algorytm 2b
2:    for i := 1 to n do begin
3:      for j := 1 to n do c[j] := −1;
4:      j := i; q := 0;
5:      t := ∅;
6:      while (c[j] = −1) and (q < m) do begin
7:        t.insert(j);
8:        c[j] := q;
9:        j := f[j]; q := q + 1;
10:     end
11:     if q = m then r[i] := j
12:     else begin
13:       p := c[j];
14:       r[i] := t[p + ((m − q) mod (q − p)) + 1];
15:     end
16:   end
17:   return r;
```

Metoda 2c: złożoność czasowa $O(n \log m)$

Redukcja początkowej złożoności czasowej $O(n \cdot m)$ do złożoności czasowej $O(n \log m)$ ma charakter strukturalny. Potraktujmy f jako funkcję, $f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$. Interesuje nas m-krotne złożenie tej funkcji, czyli f^m. Tę wartość możemy obliczyć za pomocą szybkiego potęgowania binarnego, np. korzystając ze wzorów:

$$f^{2k+1} = f^{2k} \circ f, \quad f^{2k} = \left(f^k\right)^2.$$

Poniżej implementacja korzystająca z tego samego podejścia, ale nierekurencyjna. Warto zwrócić uwagę na to, że jej złożoność pamięciowa to $O(n)$ — przy koszcie pamięciowym rzędu $\Theta(n \log m)$ moglibyśmy nie zmieścić się w limicie pamięciowym 128 MB.

```
1:  Algorytm 2c
2:    { Oblicza złożenie g∘h funkcji reprezentowanych przez n-elementowe }
3:    { tablice g i h, wynik znajduje się w tablicy w. }
4:    function zlozenie(g, h)
5:    begin
6:      for i := 1 to n do w[i] := g[h[i]];
7:      return w;
8:    end
9:
10:   g := f;   { tablica g będzie przechowywać kolejne potęgi f^(2^a) }
11:   for i := 1 to n do r[i] := i;   { funkcja identycznościowa }
12:   while m > 0 do begin
13:     if m mod 2 = 1 then r := zlozenie(r, g);
14:     m := m div 2;
```

```
15:      g := zlozenie(g, g);
16:    end
17:    return r;
```

Metoda 2d: złożoność czasowa $O(n)$

Patrząc na ograniczenia z zadania ($n \leqslant 10^6$, $m \leqslant 10^{18}$), można zaryzykować hipotezę, że złożoność czasowa i pamięciowa Algorytmu 2c jest wystarczająca. I rzeczywiście, na zawodach rozwiązania korzystające z tego podejścia (oraz Metody 1d) uzyskiwały maksymalną punktację. Pomimo tego musimy wspomnieć o tym, że również Fazę II można wykonać w optymalnej złożoności czasowej i pamięciowej $O(n)$, i taki algorytm został zastosowany w rozwiązaniu wzorcowym. Niestety algorytm ten jest istotnie bardziej skomplikowany od Algorytmu 2c, dlatego też opisujemy go w sposób skrócony. Jest on w pewnym sensie usprawnieniem Metody 2b.

Kluczowym pomysłem jest grafowa interpretacja tablicy (funkcji) f. Niech zbiorem wierzchołków grafu skierowanego $G = (V, E)$ będzie $V = \{1, 2, \ldots, n\}$, natomiast krawędzie G niech będą indukowane przez wartości funkcji f, tzn.:

$$E = \{(i, f[i]) \ : \ i \ \in V\}.$$

Ilustracja takiego grafu znajduje się już w przykładzie w treści zadania. Aby teraz obliczyć $f^m[i]$, musimy odpowiedzieć na pytanie, jaki wierzchołek leży w grafie G na końcu ścieżki o długości m prowadzącej z wierzchołka i.

Nie bez znaczenia jest tutaj szczególna własność grafu G: z każdego wierzchołka wychodzi dokładnie jedna krawędź. Tego typu grafy niejednokrotnie pojawiały się już na zawodach Olimpiady Informatycznej, ostatnio w zadaniach *Mafia* z XV Olimpiady [15], *Skarbonki* z XII Olimpiady [12] oraz *Szpiedzy* z XI Olimpiady [11]. Z tego względu pomijamy dowód faktu, że G ma następującą, specyficzną strukturę: każda jego słabo spójna składowa[1] jest cyklem, do którego doczepiona jest pewna liczba drzew skierowanych (potencjalnie zero). Dokładniej, każde takie drzewo jest skierowane od liści do korzenia, przy czym korzeń jest jednym z wierzchołków cyklu.

Całe rozwiązanie Fazy II sprowadza się do podziału grafu na opisane cykle z podoczepianymi drzewami i rozpatrzeniu osobno jednych i drugich. Oto kolejne kroki tego rozwiązania:

1. Znajdujemy wszystkie cykle w G, stosując algorytm podobny do Algorytmu 2b.
2. W każdym cyklu (reprezentowanym w tablicy, czyli z dostępem swobodnym do elementów[2]) dla każdego wierzchołka i wyznaczamy $f^m[i]$, biorąc modulo długość cyklu.
3. Usuwamy krawędzie cykli z grafu.
4. Odwracamy wszystkie pozostałe krawędzie grafu, otrzymując nowy graf $G' = (V, E')$.

[1] Słabo spójna składowa w grafie skierowanym to zbiór wierzchołków należących do jednej spójnej składowej w grafie po usunięciu skierowań krawędzi.

[2] Można skorzystać z tablic dynamicznych (patrz np. typ `vector` w języku C++), można także tablice odpowiadające wszystkim cyklom umieścić kolejno w jednej, dużej tablicy.

5. Z każdego wierzchołka należącego do jakiegokolwiek cyklu w G uruchamiamy przeszukiwanie w głąb (DFS[3]). W trakcie przeszukiwania w pomocniczej tablicy t przechowujemy wierzchołki ze znajdujących się na stosie wywołań rekurencyjnych. Na podstawie zawartości tej tablicy odpowiadamy na zapytania o $f^m[i]$.

Ostatni z powyższych punktów wymaga dodatkowego wyjaśnienia, które można znaleźć w poniższym pseudokodzie. Korzystamy w nim z pomocniczej funkcji $w\_cyklu(i, p)$, która zwraca numer wierzchołka w cyklu zawierającym wierzchołek i, oddalonego o p krawędzi od i. Dodatkowo, $t.pop\_back()$ oznacza usunięcie ostatniego elementu z tablicy t.

```
1:  Algorytm 2d, końcowe przeszukiwanie.
2:    t := ∅;
3:    procedure dfs(i)
4:    begin
5:      if size(t) ⩾ m then r[i] := t[size(t) − m + 1]
6:      else r[i] := w_cyklu(t[1], m − size(t));
7:      t.insert(i);
8:      foreach (i, j) ∈ E' do dfs(j);
9:      t.pop_back();
10:   end
```

Widzimy, że złożoność czasowa i pamięciowa każdego z powyższych kroków 1-5, a zatem także całego rozwiązania, zależy liniowo od rozmiaru grafu G, czyli jest rzędu $O(n)$.

Rozwiązania

Najefektywniejsze z opisanych rozwiązań poszczególnych faz, czyli Metody 1d i 2d, składają się na rozwiązanie wzorcowe, o złożoności czasowej $O(n)$, zaimplementowane w plikach `zab.cpp`, `zab0.pas` oraz `zab1.cpp`. Jak już wspominaliśmy, jeszcze jedna kombinacja, a mianowicie połączenie Metod 1d oraz 2c, o łącznej złożoności czasowej $O(n \log m)$, uzyskiwała maksymalną punktację. Jej implementacje można znaleźć w plikach `zabs43.cpp` oraz `zabs430.pas`. Poza tym mamy jeszcze 14 innych możliwych kombinacji par metod, z których każda daje jakieś rozwiązanie mniej efektywne. Implementacje wszystkich tych rozwiązań można znaleźć w materiałach do zadania. W zależności od złożoności czasowej uzyskiwały one na zawodach od 10 do około 60 punktów.

Wśród rozwiązań niepoprawnych warto wspomnieć o rozwiązaniu, które błędnie rozstrzyga remisy (`zabb0.cpp`, 0 punktów), oraz o rozwiązaniu nieużywającym zmiennych całkowitych 64-bitowych (`zabb1.cpp`, 40 punktów).

Testy

Wśród przygotowanych testów wyróżniamy kilka kategorii:

[3]Patrz np. książka [21].

- *test losowy* — pozycje kamieni losujemy z rozkładem jednostajnym na pewnym przedziale,
- *długa ścieżka* — funkcja f odpowiada grafowi, w którym jest jeden cykl długości 2 oraz jedna długa ścieżka prowadząca do tego cyklu,
- *długa ścieżka z dowiązaniami* — funkcja f odpowiada grafowi, w którym jest jeden cykl długości 2 oraz jedna długa ścieżka prowadząca do tego cyklu, a do każdego wierzchołka na ścieżce dochodzi ścieżka długości 1,
- *długi cykl* — funkcja f odpowiada grafowi, w którym wszystkie wierzchołki poza dwoma należą do jednego długiego cyklu.

W poniższej tabeli znajdują się opisy testów wykorzystanych na zawodach.

Nazwa	n	k	m	Opis
zab1a.in	2	1	1	przypadek brzegowy
zab1b.in	2	1	1 000 000	przypadek brzegowy
zab1c.in	3	2	1	przypadek brzegowy
zab1d.in	100	2	20	długa ścieżka z dowiązaniami
zab2a.in	2 000	3	904 194	test losowy
zab2b.in	1 500	1	8194	długa ścieżka
zab2c.in	804	3	331097	długi cykl
zab2d.in	800	2	200	długa ścieżka z dowiązaniami
zab3a.in	4 000	20	48 010	test losowy
zab3b.in	4 000	1	667 379	długa ścieżka
zab3c.in	2 832	3	759 352	długi cykl
zab3d.in	4 000	2	100	długa ścieżka z dowiązaniami
zab4a.in	10 000	12	858 448	test losowy
zab4b.in	10 000	1	2 940	długa ścieżka
zab4c.in	5 660	3	684 782	długi cykl
zab4d.in	10 000	2	200	długa ścieżka z dowiązaniami
zab5a.in	20 000	17	115 432	test losowy
zab5b.in	20 000	19 999	976 475	test losowy
zab5c.in	20 000	1	9 767	długa ścieżka
zab5d.in	20 000	2	5 000	długa ścieżka z dowiązaniami
zab6a.in	6 000	7	824 435	test losowy
zab6b.in	80 000	79 999	399 963	test losowy
zab6c.in	80 000	1	39 231	długa ścieżka
zab6d.in	80 000	2	20 000	długa ścieżka z dowiązaniami

Nazwa	n	k	m	Opis
zab7a.in	7 000	1	$\approx 6 \cdot 10^{17}$	test losowy
zab7b.in	200 000	199 999	$\approx 3 \cdot 10^{17}$	test losowy
zab7c.in	300 000	1	94 893	długa ścieżka
zab7d.in	300 000	2	60 000	długa ścieżka z dowiązaniami
zab8a.in	8 000	7	$\approx 3 \cdot 10^{17}$	test losowy
zab8b.in	400 000	399 999	$\approx 8 \cdot 10^{17}$	test losowy
zab8c.in	500 000	1	253 543	długa ścieżka
zab8d.in	600 000	2	149 000	długa ścieżka z dowiązaniami
zab9a.in	9 000	3	$\approx 5 \cdot 10^{17}$	test losowy
zab9b.in	700 000	538 348	10^{18}	test losowy
zab9c.in	834 546	1	377 651	długa ścieżka
zab9d.in	1 000 000	2	150 000	długa ścieżka z dowiązaniami
zab10a.in	10 000	8	$\approx 4 \cdot 10^{17}$	test losowy
zab10b.in	900 000	831 505	10^{18}	test losowy
zab10c.in	1 000 000	1	498 533	długa ścieżka
zab10d.in	1 000 000	2	400 000	długa ścieżka z dowiązaniami