Marcin Kubica
Treść zadania, Opracowanie

Jakub Radoszewski
Program

OI, Etap III, dzień drugi, 03.04.2008

Stacja

W Bajtocji zakończył się pierwszy etap reformy[1] *sieci kolejowej. Owa sieć składa się z dwukierunkowych odcinków torów łączących stacje kolejowe. Żadne dwie stacje nie są połączone więcej niż jednym odcinkiem torów. Ponadto wiadomo, że z każdej stacji kolejowej da się dojechać do każdej innej dokładnie jedną trasą. Trasa może być złożona z kilku odcinków torów, ale nigdy nie przechodzi przez żadną stację więcej niż raz.*

Celem drugiego etapu reformy jest zaplanowanie połączeń kolejowych. Bajtazar liczy na to, że mu w tym pomożesz. Aby uprościć zadanie, Bajtazar postanowił, że:

- *jedna ze stacji stanie się wielkim węzłem kolejowym i otrzyma nazwę Bitowice,*
- *ze wszystkich pozostałych stacji uruchomione zostaną połączenia kolejowe do Bitowic i z powrotem,*
- *każdy pociąg będzie jeździł między Bitowicami i drugą stacją końcową po jedynej możliwej trasie, zatrzymując się na wszystkich mijanych stacjach.*

Pozostaje pytanie o to, która stacja powinna zostać Bitowicami. Postanowiono, że system połączeń powinien być tak zaplanowany, by średni koszt przejazdu między dwiema różnymi stacjami kolejowymi był minimalny. W Bajtocji obowiązują wyłącznie bilety jednorazowe w cenie 1 *bajtalara, upoważniające do przejazdu jednym połączeniem na dowolną odległość. Tak więc koszt przejazdu między dwiema konkretnymi stacjami to minimalna liczba połączeń, jakie trzeba wykorzystać, aby przejechać z jednej stacji do drugiej.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis sieci kolejowej w Bajtocji,*
- *wyznaczy stację, która powinna zostać Bitowicami,*
- *wypisze wynik na standardowe wyjście.*

Wejście

W pierwszym wierszu wejścia zapisana jest jedna liczba całkowita n *(*$2 \leqslant n \leqslant 1\,000\,000$*) — jest to liczba stacji kolejowych. Stacje kolejowe są ponumerowane od* 1 *do* n*. Stacje łączy* $n-1$ *odcinków torów kolejowych. Są one opisane w kolejnych* $n-1$ *wierszach, po jednym w wierszu. W każdym z nich są zapisane dwie dodatnie liczby całkowite* a *oraz* b *(*$1 \leqslant a < b \leqslant n$*), oddzielone pojedynczym odstępem i oznaczające numery stacji, które łączy dany odcinek torów.*

[1] *Został on opisany w zadaniu* **Koleje** *z III etapu XIV OI [14]. Znajomość zadania Koleje nie jest jednak w najmniejszym stopniu potrzebna do rozwiązania niniejszego zadania.*

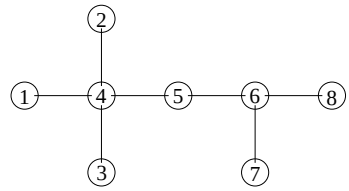
Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program powinien wypisać jedną liczbę całkowitą — optymalną lokalizację stacji Bitowice. Jeżeli istnieje więcej niż jedna optymalna odpowiedź, Twój program powinien wypisać dowolną z nich.

Przykład

Dla danych wejściowych:

```
8
1 4
5 6
4 5
6 7
6 8
2 4
3 4
```



poprawnym wynikiem jest:

```
7
```

Kółka na rysunku reprezentują stacje (liczby w kółkach to numery stacji), a krawędzie to odcinki torów. Możliwymi optymalnymi lokalizacjami Bitowic są stacje 7 oraz 8. W przypadku wyboru dowolnej z nich, średni koszt przejazdu między różnymi stacjami będzie równy $\frac{36}{28} \approx 1.2857$ *(w przykładzie jest* 28 *par nieuporządkowanych różnych stacji).*

Rozwiązanie

Analiza problemu i pierwsze pomysły na rozwiązanie

Zadanie polega na takim umiejscowieniu Bitowic, aby średni koszt przejazdu między dwiema stacjami był jak najmniejszy. Mamy n stacji, a więc jest $\frac{n \cdot (n-1)}{2}$ (nieuporządkowanych) par różnych stacji. Dla uproszczenia rozważań, zamiast minimalizować średni koszt, możemy minimalizować sumę kosztów przejazdu między wszystkimi parami różnych stacji. Koszt przejazdu między każdymi dwiema stacjami wynosi:

- 1 — jeśli istnieje między nimi bezpośrednie połączenie,
- 2 — jeśli bezpośrednie połączenie nie istnieje, to zawsze można dojechać z jednej stacji do drugiej z jedną przesiadką (np. w Bitowicach).

Naszym celem jest zatem takie umiejscowienie Bitowic, aby było jak najwięcej par stacji, między którymi istnieje bezpośrednie połączenie.

Sieć połączeń kolejowych Bajtocji jest *spójna* (między każdymi dwiema stacjami istnieje połączenie) i mamy $n-1$ odcinków torów łączących n stacji — oznacza to, że sieć połączeń kolejowych tworzy *drzewo*. Wyobraźmy sobie, że za korzeń tego drzewa wybieramy właśnie Bitowice. Po ukorzenieniu drzewa możemy mówić, że jedne wierzchołki są przodkami (lub potomkami) innych. Zauważmy, że koszt przejazdu między dwiema stacjami:

- jest równy 1, gdy jedna ze stacji jest przodkiem drugiej;
- w pozostałych przypadkach koszt wynosi 2.

Algorytm $O(n^4)$. Proste rozwiązanie może więc polegać na sprawdzeniu każdej lokalizacji Bitowic i przeszukaniu drzewa dla każdej pary stacji, w celu określenia ich wzajemnego położenia (czy jedna jest przodkiem drugiej). Niestety, jest to strasznie nieefektywny algorytm, działający w czasie $O(n^4)$. Jego implementacja znajduje się w pliku `stas2.cpp`.

Pierwszy algorytm $O(n^3)$. Oznaczmy przez $\delta(v,w)$ odległość między wierzchołkami v i w. Zauważmy, że w jest przodkiem v wtedy i tylko wtedy, gdy:

$$\delta(v, Bitowice) = \delta(v,w) + \delta(w, Bitowice)$$

Spostrzeżenie to można wykorzystać, konstruując następujący algorytm. Wystarczy raz obliczyć odległości między wszystkimi parami wierzchołków (za pomocą algorytmu Floyda-Warshalla[1], w czasie $O(n^3)$). Następnie dla każdej lokalizacji Bitowic można rozważyć każdą parę stacji, sprawdzając dla nich w czasie stałym powyższą równość. Daje to rozwiązanie działające w czasie $O(n^3)$ i wymagające pamięci rzędu $O(n^2)$, co niestety nadal nie jest akceptowalne przy podanym w zadaniu ograniczeniu na n.

Postawmy problem trochę inaczej

Poszukując dalszych ulepszeń, zastanówmy się, ile jest takich par różnych stacji, że koszt przejazdu między nimi jest równy 1? Zamiast liczyć, jak dotychczas, pary nieuporządkowane, możemy równie dobrze liczyć pary uporządkowane wierzchołków (v,w), gdzie $v \neq w$ i w jest przodkiem v. Ile jest takich par dla ustalonego v? Otóż dokładnie tyle, jaka jest odległość v od korzenia, czyli od Bitowic. Naszym celem jest więc takie umiejscowienie Bitowic, aby:

suma odległości Bitowic od wszystkich wierzchołków była jak największa.

To oznacza, że chcemy zmaksymalizować następującą sumę:

$$\sum_{v=1}^{n} \delta(Bitowice, v).$$

Drugi algorytm $O(n^3)$. Każdy składnik sumy potrafimy obliczyć w czasie $O(n)$, przeszukując sieć połączeń. Niestety, składników jest n, więc dla danego położenia Bitowic wartość sumy zostanie obliczona w czasie $O(n^2)$. Testowanie wszystkich lokalizacji Bitowic, w celu wyznaczenia optymalnej, doprowadza więc do algorytmu działającego w czasie $O(n^3)$ i pamięci $O(n)$. Jego implementacja znajduje się w pliku `stas3.cpp`.

[1] Opis tego algorytmu można znaleźć w podstawowych podręcznikach z algorytmiki, na przykład w [15] i [20].

Algorytm $O(n^2)$. Chwila namysłu pozwala nam wprowadzić do rozwiązania małą poprawkę, dającą w efekcie sporą oszczędność czasu. Żeby wyznaczyć wartość podanej sumy dla ustalonego położenia Bitowic, wystarczy tylko raz przeszukać sieć połączeń (wszerz lub w głąb) i w trakcie tego przeszukiwania wyznaczyć najkrótsze odległości od Bitowic do wszystkich wierzchołków — można to zrobić w czasie liniowym. Sprawdzenie wszystkich możliwych lokalizacji Bitowic możemy w ten sposób wykonać w czasie $O(n^2)$ i pamięci $O(n)$. Takie rozwiązanie zostało zaimplementowane w pliku `stas0.cpp`.

Algorytm $O(n^2)$ z poprawką. Kolejny moment refleksji i dochodzimy do wniosku, że nie musimy sprawdzać wszystkich możliwych lokalizacji Bitowic. Następujący fakt pozwala nam ograniczyć się w rozważaniach tylko do liści.

Fakt 1. *Optymalna lokalizacja Bitowic musi znajdować się w liściu drzewa, które tworzy sieć połączeń.*

Dowód: Załóżmy przeciwnie, że optymalna lokalizacja Bitowic znajduje się w wierzchołku wewnętrznym. Tak więc Bitowice znajdują się w *punkcie artykulacji* — wierzchołku, którego usunięcie podzieliłoby sieć połączeń na kilka (przynajmniej dwie) spójnych składowych. Oznaczmy przez $m_1, m_2, \ldots, m_k$ liczby wierzchołków w tych składowych. Bez straty ogólności możemy przyjąć, że $m_1 \leqslant m_2 \leqslant \cdots \leqslant m_k$. Przesuwając Bitowice o jedną krawędź w stronę pierwszej (tj. najmniejszej) z tych składowych, zwiększylibyśmy sumę odległości z Bitowic do wszystkich pozostałych wierzchołków o $m_2 + m_3 + \cdots + m_k - m_1 + 1 > 0$ (zagadka dla Czytelnika: dlaczego zachodzi ta nierówność?). Jest to jednak sprzeczne z założeniem, że ulokowaliśmy Bitowice optymalnie. ■

Rozwiązania korzystające z tej obserwacji zostały zaimplementowane w plikach `stas1.cpp` i `stas4.pas`. Niestety, liczba liści w drzewie może być liniowa ze względu na liczbę wszystkich wierzchołków. Tak więc czas działania tych rozwiązań nadal musimy szacować na $O(n^2)$.

Kompleksowe badanie lokalizacji opłaca się — algorytm liniowy

Spróbujmy podejść do problemu bardziej kompleksowo i wysnuć wnioski, które można wykorzystać przy szacowaniu jakości wielu lokalizacji Bitowic jednocześnie. Wartością, którą można stosunkowo szybko obliczyć dla wszystkich wierzchołków drzewa (przekonamy się, że w czasie liniowym) i która wystarcza do wyznaczenia optymalnej lokalizacji Bitowic, jest

suma odległości od danego wierzchołka do wszystkich pozostałych.

Wówczas problem sprowadza się do wybrania wierzchołka o największej sumie odległości i uzyskujemy rozwiązanie działające w czasie $O(n)$.

Wyznaczmy na korzeń drzewa stanowiącego sieć połączeń dowolny wierzchołek, na przykład ten o numerze 1. Dwukrotnie przeglądając rekurencyjnie drzewo połączeń,

obliczamy dla każdego wierzchołka poszukiwaną sumę odległości do wszystkich pozostałych wierzchołków.
Najpierw, dla każdego wierzchołka v wyznaczamy:

- l_v — liczbę wierzchołków w poddrzewie o korzeniu w v oraz
- s_v — sumę odległości od v do wierzchołków w poddrzewie o korzeniu w v.

Wartości te obliczamy, idąc w górę drzewa, od liści do korzenia. Dla liści jest to proste:

$$l_v = 1, \;\; s_v = 0.$$

Jeśli v nie jest liściem, to niech $v_1, v_2, \ldots, v_k$ będą jego następnikami (synami). Wówczas wartości $l_{v_1}, \ldots, l_{v_k}$ oraz $s_{v_1}, \ldots, s_{v_k}$ zostały już obliczone i zachodzą zależności:

$$l_v = l_{v_1} + l_{v_2} + \cdots + l_{v_k} + 1$$

oraz

$$\begin{aligned} s_v &= s_{v_1} + l_{v_1} + s_{v_2} + l_{v_2} + \cdots + s_{v_k} + l_{v_k} = \\ &= s_{v_1} + s_{v_2} + \cdots + s_{v_k} + l_v - 1. \end{aligned}$$

Po wyznaczeniu wartości l_v i s_v powtórnie przechodzimy rekurencyjnie drzewo. Idąc od korzenia do liści, obliczamy dla każdego wierzchołka v:

- r_v — sumę odległości od niego do wszystkich pozostałych wierzchołków.

Dla korzenia ($v = 1$) jest łatwo:

$$r_v = s_v.$$

Jeśli v nie jest korzeniem, to niech o będzie poprzednikiem (ojcem) v. Wartość r_o została już obliczona. Zastanówmy się, o ile r_v różni się od r_o? Gdy przesuwamy się z o do v, to odległość do wszystkich wierzchołków w poddrzewie o korzeniu w v zmniejsza się o 1, natomiast odległość do wszystkich pozostałych wierzchołków (wliczając o) zwiększa się o 1. Tak więc

$$r_v - r_o = -l_v + (n - l_v) = n - 2 \cdot l_v$$

czyli

$$r_v = r_o + n - 2 \cdot l_v.$$

Potrafimy więc obliczyć wszystkie wartości r_v, a tym samym wyznaczyć optymalną lokalizację Bitowic, w liniowym czasie i pamięci. Jest to rozwiązanie wzorcowe, którego implementacja znajduje się w plikach `sta.cpp`, `sta0.pas` i `sta1.java`.

Testy

Zostało przygotowanych 10 grup testów. Większość z nich to testy losowe. Są one wystarczające do sprawdzenia zarówno poprawności, jak i wydajności rozwiązań. W każdym z nich drzewo połączeń ma około $\frac{n}{2}$ liści, a wartości wyniku są raczej przypadkowe. Dodatkowo, zadbano, by w testowych sieciach połączeń liczba optymalnych lokalizacji Bitowic była niewielka.

Rozwiązania działające w czasie $O(n^4)$ i $O(n^3)$ otrzymywały 20% punktów. Rozwiązania działające w czasie $O(n^2)$ otrzymywały 40%–50% punktów, w zależności od tego, czy uwzględniały spostrzeżenie przedstawione w Fakcie 1, czy też nie. Rozwiązania działające w czasie liniowym otrzymywały, oczywiście, 100% punktów.

Poniższa tabelka zawiera zbiorcze zestawienie testów. Liczbę stacji oznaczono w niej przez n, a liczbę liści — przez l.

Nazwa	n	l	**Opis**
sta0.in	8	5	test przykładowy z treści zadania
sta1a.in	15	9	test losowy
sta1b.in	2	2	jedna krawędź
sta2.in	100	54	test losowy
sta3.in	700	343	test losowy
sta4.in	2 000	1 000	test losowy
sta5.in	500 000	7	bardzo długa ścieżka z kilkoma dodatkami
sta6.in	40 000	19 963	test losowy
sta7.in	100 000	49 973	test losowy
sta8a.in	250 000	124 919	test losowy
sta8b.in	200 001	100 000	gwiazda o ramionach długości 2
sta9a.in	500 000	250 092	test losowy
sta9b.in	800 001	4	gwiazda o ramionach długości 200 000
sta10a.in	1 000 000	499 693	test losowy
sta10b.in	1 000 000	7	bardzo długa ścieżka z kilkoma dodatkami

Jakub Radoszewski
Treść zadania, Opracowanie

Wojciech Tyczyński
Program

OI, Etap III, dzień drugi, 03.04.2008

Permutacja

Multizbiorem nazywamy obiekt matematyczny podobny do zbioru, w którym jednak ten sam element może występować wielokrotnie. Podobnie jak w przypadku zbioru, także dla multizbioru wszystkie elementy można ustawić w ciąg i to zazwyczaj na wiele sposobów; każde takie ustawienie nazywamy **permutacją** *multizbioru. Dla przykładu, permutacjami multizbioru* $\{1,1,2,3,3,3,7,8\}$ *są między innymi* $(2,3,1,3,3,7,1,8)$ *oraz* $(8,7,3,3,3,2,1,1)$.

Powiemy, że jedna permutacja danego multizbioru jest mniejsza (w porządku leksykograficznym) od drugiej, jeżeli na pierwszej pozycji, na której te permutacje się różnią, pierwsza z nich zawiera element mniejszy niż druga. Wszystkie permutacje **danego multizbioru** *można ponumerować (zaczynając od jedynki) w kolejności od najmniejszej do największej.*

Zadanie

Napisz program, który:

- *wczyta ze standardowego wejścia opis permutacji pewnego multizbioru oraz dodatnią liczbę* m,
- *wyznaczy numer wczytanej permutacji w porządku leksykograficznym, a dokładniej jego resztę z dzielenia przez* m,
- *wypisze wynik na standardowe wyjście.*

Wejście

Pierwszy wiersz wejścia zawiera dwie liczby całkowite n *oraz* m $(1 \leqslant n \leqslant 300\,000,\ 2 \leqslant m \leqslant 1\,000\,000\,000)$, *oddzielone pojedynczym odstępem. Oznaczają one odpowiednio liczbę elementów multizbioru oraz ... liczbę* m. *Drugi wiersz wejścia zawiera* n *dodatnich liczb całkowitych* a_i $(1 \leqslant a_i \leqslant 300\,000)$, *pooddzielanych pojedynczymi odstępami i oznaczających kolejne elementy danej permutacji multizbioru.*

Wyjście

Pierwszy i jedyny wiersz wyjścia powinien zawierać jedną liczbę całkowitą, oznaczającą resztę z dzielenia przez m *numeru podanej permutacji w porządku leksykograficznym.*

Przykład

Dla danych wejściowych:

```
4 1000
2 1 10 2
```

poprawnym wynikiem jest:

```
5
```

Wszystkie permutacje mniejsze od zadanej to (w kolejności leksykograficznej): $(1,2,2,10)$, $(1,2,10,2)$, $(1,10,2,2)$ oraz $(2,1,2,10)$.

Rozwiązanie

Ile tego jest?

Zadania polega na wyznaczeniu numeru zadanej permutacji $a = a_1, a_2, \ldots, a_n$ pewnego multizbioru w porządku leksykograficznym. Na początek zastanówmy się, ile w ogóle jest różnych permutacji multizbioru $Z = \{a_1, a_2, \ldots, a_n\}$, czyli jak duży może być wynik.

Niech $z = \max\{a_1, \ldots, a_n\}$, wówczas $Z \subseteq \{1, 2, \ldots, z\}$ (w tym zapisie traktujemy Z jako zbiór). Przez $l_1, \ldots, l_z$ oznaczmy krotności (liczby wystąpień) liczb $1, \ldots, z$ w multizbiorze Z. Liczba wszystkich permutacji n-elementowego *zbioru* to $n!$, czyli dla przypomnienia:

$$\begin{aligned} 0! &= 1, \\ n! &= n \cdot (n-1)! \quad \text{dla } n \geqslant 1. \end{aligned}$$

Multizbiór Z możemy przekształcić w zbiór, numerując wystąpienia poszczególnych liczb w Z, czyli zastępując kolejne wystąpienia elementu $i \in \{1, \ldots, z\}$ w Z parami uporządkowanymi:

$$(i,1),\ (i,2),\ \ldots,\ (i,l_i).$$

Utworzony w ten sposób zbiór oznaczmy przez Z'. Liczba permutacji Z' to oczywiście $n!$.

Ilu permutacjom zbioru Z' odpowiada jedna permutacja p multizbioru Z? Wystąpienia liczby 1 w p można ponumerować od 1 do l_1 na $l_1!$ sposobów, co za każdym razem da w rezultacie inną permutację Z'. Podobnie, wystąpienia kolejnych liczb $2, \ldots, z$ można ponumerować odpowiednio na $l_2!$, $\ldots$, $l_z!$ sposobów, co daje łączną liczbę różnych permutacji Z', jakie odpowiadają p, równą $l_1! \cdot l_2! \cdot \ldots \cdot l_z!$. Zatem liczba różnych permutacji multizbioru Z to

$$\frac{n!}{l_1! \cdot l_2! \cdot \ldots \cdot l_z!}. \tag{1}$$

Jak się za to zabrać?

Numer permutacji a w porządku leksykograficznym możemy wyznaczyć wprost — wystarczy przejrzeć wszystkie permutacje i zliczyć te, które są mniejsze od niej. Musimy przy tym zadbać, by żadnej nie policzyć wielokrotnie. Realizacja tego pomysłu wymaga przejrzenia $O(n!)$ permutacji i wykonania dla każdej z nich porównania z a w czasie $O(n)$. Daje to rozwiązanie o złożoności czasowej $O(n \cdot n!)$, które zaimplementowaliśmy w pliku `pers0.cpp`.

Dla $n = 10$ liczba elementarnych operacji takiego prostego algorytmu może wynieść około 30 milionów. To być może jeszcze nie nadużyje cierpliwości sprawdzających, ale dla $n = 11, 12, \ldots$ Rzut oka na ograniczenia z zadania wystarcza, aby zauważyć, że musimy wymyślić coś znacznie, ale to znacznie szybszego.

Poszukiwanie bardziej efektywnego rozwiązania rozpoczniemy od wymyślenia sposobu systematycznego przeanalizowania wszystkich permutacji mniejszych od zadanej permutacji

a. Dowolna permutacja b mniejsza od a jest na pewnym prefiksie (tj. początkowym fragmencie) długości $i-1$ zgodna z a, następnie znajduje się w niej element $b_i < a_i$, po którym następuje całkiem już dowolny ciąg elementów.

Definicja 1. Powiemy, że permutacja b jest *kategorii* i względem permutacji a, jeżeli

$$(\forall j < i)\ (b_j = a_j) \text{ oraz } b_i < a_i.$$

W ten sposób uzasadniliśmy następujący fakt:

Fakt 1. *Każda permutacja multizbioru Z mniejsza leksykograficznie od a należy do dokładnie jednej kategorii $i \in \{1,\ldots,n\}$ względem a.*

Skupmy się więc na odrębnym zliczeniu permutacji należących do każdej z kategorii $1,2,\ldots,n$. Warto przy tym zauważyć jeszcze następujący fakt:

Fakt 2. *Permutacje przynależne do kategorii i możemy utożsamiać z permutacjami kategorii* 1 *względem permutacji*

$$p_i = a_i, a_{i+1}, \ldots, a_n$$

multizbioru

$$Z_i = \{a_i, a_{i+1}, \ldots, a_n\}.$$

Permutacje kategorii i

Niech $l_{i,1}, l_{i,2}, \ldots, l_{i,z}$ oznaczają krotności liczb $1,2,\ldots,z$ w multizbiorze Z_i (dla $1 \leqslant i \leqslant n$). Wówczas każda permutacja b należąca do kategorii 1 względem p_i zaczyna się od pewnego elementu $j < a_i$, takiego że $l_{i,j} > 0$; po nim następuje jakakolwiek permutacja multizbioru $\{a_i, a_{i+1}, \ldots, a_n\} \setminus \{j\}$. Z tego wynika, że liczba wszystkich permutacji b kategorii i względem a wyraża się wzorem

$$K_i = \sum_{j<a_i, l_{i,j}>0} \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,j-1}! \cdot (l_{i,j}-1)! \cdot l_{i,j+1}! \cdot \ldots \cdot l_{i,z}!}. \tag{2}$$

Delikatnie rzecz ujmując, powyższy wzór na K_i wygląda koszmarnie. Już pobieżna analiza pozwala zauważyć, że wyliczanie sumy $K_1 + K_2 + \ldots + K_n \pmod m$ poprzez bezpośrednie zastosowanie tego wzoru będzie wymagać czasu $\Omega(n \cdot z^2)$. Choć jest już lepiej niż poprzednio, to jednak powinniśmy jeszcze znacznie poprawić tę metodę.

Natychmiastowe usprawnienie otrzymamy, przekształcając nieco wzór (2). Zauważmy mianowicie, że iloczyn z mianownika jest w gruncie rzeczy równy

$$(l_{i,1}! \cdot l_{i,2}! \cdot \ldots \cdot l_{i,z}!)/l_{i,j}.$$

Wykorzystując tę równość, a także wyciągając ułamek niezależny od j przed znak sumy, przekształcamy wzór na K_i do postaci:

$$K_i = \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!} \cdot \sum_{j<a_i, l_{i,j}>0} l_{i,j}.$$

Zauważmy wreszcie, że w nowej postaci wzoru warunek $l_{i,j} > 0$ jest zbędny, więc ostatecznie uzyskujemy istotnie uproszczony wzór:

$$K_i = \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!} \cdot \sum_{j=1}^{a_i-1} l_{i,j}. \tag{3}$$

Zliczanie permutacji kategoriami

Rozwiązanie wzorcowe oprzemy na wzorze (3). Na jego podstawie będziemy wyznaczać kolejno wartości $K_1, K_2, \ldots, K_n$. Postaramy się przy tym korzystać zawsze z wcześniej wyznaczonych wartości K_i, gdyż bezpośrednie zastosowanie wzoru nadal wygląda na procedurę zbyt czasochłonną dla podanych w zadaniu ograniczeń. Pierwszym krokiem do sukcesu jest znalezienie związku między wartościami $l_{i,*}$ a $l_{i+1,*}$. Korzystając z zależności $Z_{i+1} = Z_i \setminus \{a_i\}$, widzimy, że jest on bardzo prosty:

$$l_{i+1,k} = \begin{cases} l_{i,k} - 1 & \text{jeżeli } k = a_i, \\ l_{i,k} & \text{w przeciwnym przypadku.} \end{cases} \tag{4}$$

Następnie zauważmy, że wzór (3) składa się z dwóch niezależnych czynników, które możemy wyznaczyć niezależnie: ułamka

$$U_i = \frac{(n-i)!}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!}$$

i sumy

$$S_i = \sum_{j=1}^{a_i-1} l_{i,j}.$$

Ułamki U_i

Zależność (4) między wartościami $l_{i,*}$ a $l_{i+1,*}$ pozwala następująco przekształcić wzór na U_{i+1}:

$$\begin{aligned} U_{i+1} &= \frac{(n-(i+1))!}{l_{i+1,1}! \cdot \ldots \cdot l_{i+1,z}!} \\ &= \frac{(n-i)!/(n-i-1)}{l_{i,1}! \cdot \ldots \cdot l_{i,z}!/l_{i,a_i}} \\ &= \frac{l_{i,a_i}}{n-i-1} \cdot U_i \end{aligned}$$

Po tym przekształceniu widzimy, że wystarczy tylko U_1 obliczyć wprost ze wzoru:

$$U_1 = \frac{(n-1)!}{l_{1,1}! \cdot \ldots \cdot l_{1,z}!} = \frac{(n-1)!}{l_1! \cdot \ldots \cdot l_z!},$$

po czym każde kolejne U_{i+1} możemy policzyć, mnożąc U_i przez $l_{i,a_i}/(n-i-1)$. Pomijając na razie problem pojawiających się w obliczeniach dużych liczb, daje to już całkiem efektywną metodę, wymagającą wykonania łącznie $O(n+z)$ mnożeń i dzieleń (zauważmy, że $l_1 + \ldots + l_z = n$).

Uwaga 1. Jakkolwiek ułamki U_i mogą być niecałkowite, to iloczyny $S_i \cdot U_i$ już całkowite być muszą, a zatem przy zachowaniu w algorytmie odpowiedniej kolejności mnożeń i dzieleń można przy wszystkich obliczeniach pozostać w dziedzinie liczb całkowitych.

Sumy S_i

Wyznaczanie kolejnych wartości S_i może okazać się prostsze, gdy przyjrzymy się, jakie jest ich znaczenie. Przypomnijmy:

$$Z_i = \{a_i, a_{i+1}, \ldots, a_n\} \text{ oraz } Z_{i+1} = \{a_{i+1}, \ldots, a_n\}.$$

Wartość S_i to liczba elementów mniejszych od a_i w multizbiorze Z_i. Analogicznie S_{i+1} to liczba elementów mniejszych od a_{i+1} w multizbiorze Z_{i+1}. Jeśli umieścimy zbiór Z_i w strukturze, dzięki której łatwo odpowiedzieć na pytanie o liczbę elementów mniejszych od dowolnej zadanej wartości x, to aby wyznaczyć S_{i+1} musimy:

- usunąć element a_i ze zbioru — dostajemy automatycznie strukturę ze zbiorem Z_{i+1};
- zapytać o liczbę elementów mniejszych od a_{i+1} w strukturze.

Do implementacji powyższej struktury świetnie nadaje się statyczne drzewo binarne, zwane *drzewem licznikowym* albo *przedziałowym*. Ma ono z liści, które odpowiadają kolejno (od lewej) wartościom $1, 2, \ldots, z$. Gdy rozważamy zbiór Z_i, to w j-tym liściu mamy wartość $l_{i,j}$, czyli liczbę wystąpień elementu j w zbiorze Z_i. W każdym węźle wewnętrznym v drzewa przechowujemy sumę wartości z wszystkich liści poddrzewa ukorzenionego w v, czyli liczbę elementów ze zbioru $[j_1, j_2] \cap Z_i$, gdzie j_1 jest najmniejszym, a j_2 największym liściem w poddrzewie v[1]. Koszt czasowy konstrukcji, a zarazem rozmiar takiego drzewa dla ciągu z-elementowego to $O(z)$. Natomiast każdą z potrzebnych operacji możemy z jego pomocą wykonać w czasie $O(\log z)$. To daje łączną złożoność czasową wyznaczenia wszystkich sum S_i rzędu $O(z + n \cdot \log z)$.

Działania modulo

Mogłoby się wydawać, że opracowanie efektywnych metod wyznaczania ułamków U_i oraz sum S_i to już koniec zadania. Byłoby tak, gdyby wszystkie liczby, z którymi mamy do czynienia, były niewielkie i całkowite. Niestety liczby K_i nie są małe a w obliczeniach pojawiają się ułamki U_i.

Ponieważ ostatecznie musimy podać wynik modulo m, więc jedyne, czego nam brakuje, to umiejętność wykonywania modulo m wszystkich działań pośrednich, czyli głównie mnożenia i dzielenia[2]. Przypomnijmy, że w działaniach modulo m ostatecznie wszystkie liczby sprowadzamy do reszt z dzielenia całkowitego przez m, czyli do wartości $\{0, 1, \ldots, m-1\}$. Zapis $a \bmod m$ oznacza właśnie resztę z dzielenia a przez m. Piszemy także, że $a \equiv b \pmod{m}$, jeżeli reszty z dzielenia a przez m oraz b przez m są takie same.

[1] Więcej o statycznych drzewach przedziałowych można przeczytać np. w opisie rozwiązania zadania Tetris 3D z książeczki XIII Olimpiady Informatycznej [13].

[2] Więcej o działaniu modulo i o przystawaniu (kongruencji) modulo można przeczytać w podstawowych podręcznikach z matematyki dyskretnej: [31] i [32], a także w [20].

Samo mnożenie nie stwarza problemów, gdyż, jak łatwo się przekonać, zachodzi równość:

$$(a \cdot b \cdot c) \bmod m = \big(((a \cdot b) \bmod m) \cdot c\big) \bmod m.$$

Oznacza to, że po każdym mnożeniu wynik można „zredukować" do reszty z dzielenia przez m. Dużo gorzej wygląda sytuacja z dzieleniem. Aby przekonać się o tym, wystarczy przyjrzeć się kilku przykładom:

- ponieważ $7 \cdot 5 = 35 \equiv 11 \pmod{12}$, to $11/5 \equiv 7 \pmod{12}$;
- jako że $2 \cdot 4 = 8$, ale także $8 \cdot 4 = 32 \equiv 8 \pmod{12}$, zatem wynik dzielenia $8/4$ na resztach modulo 12 nie jest jednoznaczny;
- nie istnieje żadna liczba naturalna x, dla której zachodziłoby $x \cdot 6 \equiv 1 \pmod{12}$, dlatego dzielenie $1/6$ nie jest wykonalne na resztach modulo 12.

Widać, że sytuacja wygląda raczej niewesoło — trudno wykonać dzielenie, jeśli szukany iloraz nie istnieje albo jest niejednoznaczny. Nawet w przypadku, gdy wynik jest dobrze określony, to na pierwszy rzut oka nie bardzo widać, jak go wyznaczyć.

Ponieważ dzielenie modulo m okazuje się być takie kłopotliwe, to spróbujmy sprowadzić je do mnożenia. Dzielenie modulo, podobnie jak zwykłe dzielenie, jest operacją odwrotną do mnożenia i dzielenie przez x można interpretować jako mnożenie przez odwrotność x, czyli x^{-1}:

$$x \cdot x^{-1} \equiv 1 \bmod m,$$

$$a/x \equiv a \cdot x^{-1} \bmod m.$$

Dzięki powyższej zamianie dzielenie nie staje się automatycznie prostsze: wartość x^{-1} nie musi istnieć i nie musi być wyznaczona jednoznacznie. Jednak w pewnych przypadkach jest łatwo — okazuje się, że dla x względnie pierwszego z m odwrotność x modulo m zawsze istnieje i potrafimy ją efektywnie wyznaczyć.

Fakt 3. *Jeżeli x jest względnie pierwsze z m, to x^{-1} mod m jest określone jednoznacznie.*

Dowód tego faktu, wykorzystujący tzw. *rozszerzony algorytm Euklidesa*, można znaleźć w Dodatku na końcu niniejszego opisu rozwiązania. Jest on opisany także w książce [20]. Złożoność czasowa tego algorytmu zastosowanego do liczb $x < m$ oraz m to $O(\log m)$.

Jeszcze jeden drobny kłopot

Gdyby m było dużą liczbą pierwszą, to w trakcie obliczeń mielibyśmy do czynienia tylko z liczbami względnie pierwszymi z m i problem dzielenia modulo m byłby już rozwiązany. Niestety, nie możemy na to liczyć.

Niech

$$m = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}$$

będzie rozkładem m na czynniki pierwsze ($p_i \neq p_j$ dla $i \neq j$). Widzimy, że kłopoty przy liczeniu odwrotności mogą sprawić tylko liczby podzielne przez którąś z liczb p_i. Aby sobie

z nimi poradzić, wydzielmy z rozważanych liczb część podzielną przez liczby p_i. W tym celu wszystkie liczby w algorytmie będziemy reprezentować w postaci

$$v \cdot p_1^{a_1} \cdot \ldots \cdot p_k^{a_k},$$

gdzie $NWD(v,m) = 1$ (zwróć uwagę, że liczby pojawiające się w wykładnikach to a_i, a nie α_i). Wszystkie potrzebne nam operacje wykonujemy na tak zapisanych liczbach następująco:

- *Mnożenie* liczb wykonujemy poprzez wymnożenie współczynników v i dodanie do siebie odpowiednich wykładników potęg a_i. Koszt czasowy mnożenia to po prostu $O(k)$.

$$\left(v \cdot p_1^{a_1} \cdot \ldots \cdot p_k^{a_k}\right) \cdot \left(v' \cdot p_1^{a'_1} \cdot \ldots \cdot p_k^{a'_k}\right) = (v \cdot v' \bmod m) \cdot p_1^{a_1+a'_1} \cdot \ldots \cdot p_k^{a_k+a'_k}$$

- *Dzielenie* dwóch liczb wykonujemy, dzieląc modulo m współczynnik v pierwszej liczby przez współczynnik drugiej (co jest wykonalne, gdyż oba są względnie pierwsze z m) i odejmując wykładniki a_i drugiej liczby od wykładników pierwszej. Koszt czasowy dzielenia to zatem $O(k + \log m)$, gdzie drugi składnik wynika z konieczności użycia algorytmu Euklidesa do policzenia odwrotności modulo m.

$$\left(v \cdot p_1^{a_1} \cdot \ldots \cdot p_k^{a_k}\right) / \left(v' \cdot p_1^{a'_1} \cdot \ldots \cdot p_k^{a'_k}\right) = (v \cdot v'^{-1} \bmod m) \cdot p_1^{a_1-a'_1} \cdot \ldots \cdot p_k^{a_k-a'_k}$$

- *Przekształcenie liczby całkowitej x do postaci* $(v, a_1, \ldots, a_k)$ wykonujemy, dzieląc x kolejno przez $p_1, \ldots, p_k$. Sumarycznie operacji dzielenia wykonamy najwyżej $O(\log(n+m))$, gdyż dla liczby x suma wykładników a_i nie przekracza logarytmu z x, a będziemy mieć do czynienia tylko z x wielkości $O(\max(n,m))$. Uwzględniając, że musimy także sprawdzić każdą wartość p_i, widzimy, że złożoność czasowa tej operacji wynosi $O(k + \log(n+m))$.

- *Przekształcenie liczby zapisanej w postaci* $(v, a_1, \ldots, a_k)$ na zapisaną „normalnie" resztę modulo m wykonujemy za pomocą szybkiego potęgowania binarnego modulo m. Obliczenie tą metodą $x^y \bmod m$ wymaga $O(\log y)$ mnożeń liczb rzędu $O(m)$. Największą liczbą, jaka może pojawić się w trakcie obliczeń w naszym rozwiązaniu jest, jak łatwo sprawdzić, $n!$. Stąd mamy

$$a_i \leqslant \log_{p_i} n! \leqslant \log_2 n! = \log(1 \cdot 2 \cdot \ldots \cdot n) = \log 1 + \log 2 + \ldots + \log n \leqslant n \cdot \log n.$$

  To oznacza, że koszt czasowy całej operacji zamiany wynosi $O(k \cdot \log(n \cdot \log n)) = O(k \log n)$.

Z przeprowadzonych rozważań wynika, że każdą z potrzebnych operacji potrafimy wykonać w czasie $O(\log m \cdot \log n)$ (zauważmy, że $k = O(\log m)$). Biorąc pod uwagę ograniczenia z zadania, oraz to, że każda z tych operacji jest wykonywana w algorytmie $O(n+z)$ razy, daje to efektywne rozwiązanie problemu.

Opisane rozwiązanie wzorcowe zostało zaimplementowane w plikach `per2.cpp` oraz `per4.java`.

Alternatywna wersja rozwiązania wzorcowego

Ostatni krok rozwiązania wzorcowego, czyli sposób wykonywania działań na resztach z dzielenia przez m, można zaimplementować także trochę inaczej, z wykorzystaniem konstruktywnej wersji Chińskiego twierdzenia o resztach.

Twierdzenie 1 (Chińskie twierdzenie o resztach). *Niech $y_1, \ldots, y_k$ będą dowolnymi liczbami całkowitymi, a $n_1, \ldots, n_k$ — liczbami całkowitymi parami względnie pierwszymi. Wówczas układ*

$$\begin{aligned} x &\equiv y_1 \pmod{n_1} \\ x &\equiv y_2 \pmod{n_2} \\ &\ldots \\ x &\equiv y_k \pmod{n_k} \end{aligned} \tag{5}$$

spełnia dokładnie jedna reszta x_0 modulo $N = n_1 n_2 \cdot \ldots \cdot n_k$.

Dowód: Podobnie jak w dowodzie faktu o odwrotności modulo, i tym razem rozpoczniemy od wykazania jednoznaczności x_0. Jeżeli więc x_1 również spełnia układ kongruencji (5), to

$$(\forall\ 1 \leqslant i \leqslant k)\ (x_0 \equiv y_i \equiv x_1 \pmod{n_i}).$$

Stąd

$$(\forall\ 1 \leqslant i \leqslant k)\ (n_i \mid (x_0 - x_1)),$$

a więc

$$NWW(n_1, \ldots, n_k) \mid (x_0 - x_1),$$

gdzie NWW oznacza najmniejszą wspólną wielokrotność. Ponieważ liczby n_i są parami względnie pierwsze, więc $NWW(n_1, \ldots, n_k) = n_1 \cdot \ldots \cdot n_k$ i powyższa podzielność implikuje już żądaną jednoznaczność, gdyż jest równoważna

$$x_0 \equiv x_1 \pmod{N}.$$

Pokażmy teraz, że reszta x_0 istnieje. Jest kilka niekonstruktywnych dowodów tego faktu (wykazują istnienie x_0, ale nie wskazują, jak je wyliczyć). Na szczęście znamy także dowód konstruktywny. Zawiera on jawny wzór na x_0:

$$x_0 = \sum_{i=1}^{k} \left(\frac{N}{n_i}\right) \cdot \left(\left(\frac{N}{n_i}\right)^{-1} \bmod n_i\right) \cdot y_i \tag{6}$$

Nie bardzo widać, skąd ten wzór się wziął, ale na szczęście sprawdzenie, że jest poprawny, nie jest trudne. Rozważmy konkretną wartość $j \in \{1, \ldots, k\}$ i pokażmy, że

$$x_0 \equiv y_j \pmod{n_j}.$$

Zauważmy, że dla $i \neq j$ składniki sumy (6) są modulo n_j równe 0, gdyż $n_j \mid (N/n_i)$. Pozostaje zatem przyjrzeć się jedynie j-emu składnikowi sumy (6), czyli:

$$\left(\frac{N}{n_j}\right) \cdot \left(\left(\frac{N}{n_j}\right)^{-1} \bmod n_j\right) \cdot y_j.$$

Czynnik

$$\frac{N}{n_j} = n_1 \cdot \ldots \cdot n_{j-1} \cdot n_{j+1} \cdot \ldots \cdot n_k$$

jest względnie pierwszy z n_j, więc istnieje $(N/n_j)^{-1} \bmod n_j$ — jego odwrotność modulo n_j. Oczywiście z definicji odwrotności prawdziwa jest kongruencja

$$\left(\frac{N}{n_j}\right) \cdot \left(\left(\frac{N}{n_j}\right)^{-1} \bmod n_j\right) \equiv 1 \pmod{n_j}.$$

To pozwala zauważyć, że reszta z dzielenia j-ego wyrazu sumy (6) przez n_j wynosi

$$\left(\frac{N}{n_j}\right) \cdot \left(\left(\frac{N}{n_j}\right)^{-1} \bmod n_j\right) \cdot y_j \equiv y_j \pmod{n_j}.$$

Jest to także reszta z dzielenia x_0 przez n_j, co kończy dowód poprawności wzoru (6). ■

Jak można wykorzystać Chińskie twierdzenie o resztach do efektywnego wykonywania działań na resztach modulo m? Jak poprzednio, przedstawimy liczbę m w postaci

$$m = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k},$$

gdzie $NWD(p_i, p_j) = 1$ dla $i \neq j$. Przyjmując $n_i = p_i^{\alpha_i}$, dostajemy układ $n_1, \ldots, n_k$, w którym liczby z przedziału $[0, m-1]$ możemy reprezentować za pomocą reszt. Każdą z reszt modulo n_i możemy z kolei zapisać, jak poprzednio, w postaci $v_i \cdot p_i^{a_i}$, gdzie $v_i \in \{0, \ldots, p_i^{\alpha_i} - 1\}$ jest względnie pierwsze z p_i.

To oznacza, że reprezentacja liczby x ma wówczas postać wektora o $2k$ elementach: $(v_1, \ldots, v_k, a_1, \ldots, a_k)$. Operacje na tak zapisanych liczbach wykonujemy analogicznie jak poprzednio. Na przykład dla reszt x i y, o reprezentacjach:

$$x \to (v_1, \ldots, v_k, a_1, \ldots, a_k),$$

$$y \to (v'_1, \ldots, v'_k, a'_1, \ldots, a'_k),$$

reprezentacja ich iloczynu jest równa

$$(x \cdot y) \to ((v_1 \cdot v'_1) \bmod p_1^{\alpha_1}, \ldots, (v_k \cdot v'_k) \bmod p_k^{\alpha_k}, a_1 + a'_1, \ldots, a_k + a'_k).$$

Dzielenie i przekształcanie liczby ze „zwykłego” zapisu do omawianej reprezentacji odbywa się analogicznie, jak poprzednio. Natomiast przekształcenie odwrotne jest możliwe dzięki Chińskiemu twierdzeniu o resztach, konkretnie dzięki wzorowi (6).

Można by się zapytać, po co właściwie zamieszczony został opis tej wersji sposobu reprezentacji reszt modulo m, skoro wydaje się ona bardzo podobna do tej z rozwiązania wzorcowego, a przy tym jeszcze nieco bardziej skomplikowana. Pierwszym powodem jest fakt, że przy zamianie nowej reprezentacji liczb na resztę modulo m nie potrzeba szybkiego potęgowania modulo, gdyż jeżeli którykolwiek z wykładników a_i jest większy od odpowiadającego wykładnika α_i, to $x \equiv 0 \pmod{p_i^{\alpha_i}}$, więc nie musimy się przejmować dużymi wartościami a_i. Drugim zaś powodem było właśnie to, że Chińskie twierdzenie o resztach jest interesujące samo w sobie...

Po implementacje tej wersji rozwiązania wzorcowego odsyłamy do plików `per.cpp`, `per1.pas` oraz `per3.java`.

Rozwiązania wolne

Prawie każdy krok rozumowania prowadzący do rozwiązania wzorcowego może zostać zrealizowany w inny, mniej efektywny sposób, co prowadzi do całej gamy rozwiązań wolniejszych, które na zawodach uzyskiwały od 10% do około 55% punktów. Zainteresowanych Czytelników odsyłamy do implementacji w plikach `pers1.cpp`-`pers6.pas`.

Testy

Rozwiązania zawodników były sprawdzane na zestawie 15 testów. W większości z nich konkretne permutacje były generowane w sposób całkowicie losowy; w niektórych (testy 3, 4, 5, 8, 9 i 10) permutacje składają się z niedużej liczby szczytów, czyli fragmentów rosnąco-malejących.

Nazwa	n	m	z	Opis
per1.in	8	$2^2 \cdot 7 \cdot 11$	5	test poprawnościowy
per2.in	10	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	456	test poprawnościowy
per3.in	73	90089	995	kilka szczytów
per4.in	81	$9221 \cdot 56477$	163913	kilka szczytów
per5.in	87	2^{19}	289633	kilka szczytów
per6.in	728	$53 \cdot 193 \cdot 293$	4999	test losowy
per7.in	2741	$2^4 \cdot 3^2 \cdot 5 \cdot 7^3$	99973	test losowy
per8.in	5363	$2 \cdot 3^2 \cdot 7 \cdot 29 \cdot 293$	29481	kilka szczytów
per9.in	8179	$11 \cdot 17 \cdot 19 \cdot 29 \cdot 41$	299840	kilka szczytów
per10.in	67528	$97 \cdot 107 \cdot 94597$	148545	kilka szczytów
per11.in	100000	$28669 \cdot 28687$	9841	test losowy
per12.in	213819	$103 \cdot 163 \cdot 167 \cdot 173$	6152	test losowy
per13.in	284315	975261919	150000	test losowy
per14.in	300000	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23$	299997	test losowy
per15.in	300000	$2^9 \cdot 5^9$	300000	test losowy

Dodatek — dowód Faktu 3

Dowód rozpoczniemy nietypowo — pokażemy, że liczba x względnie pierwsza z m ma najwyżej jedną odwrotność modulo m, a dopiero później uzasadnimy, dlaczego ją w ogóle ma. Załóżmy, że y i y' spełniają równość modulo:

$$x \cdot y \equiv 1 \equiv x \cdot y' \pmod{m}.$$

Wówczas także

$$m \mid xy - xy' = x(y - y').$$

Ponieważ x i m są względnie pierwsze, to ta podzielność oznacza, że

$$m \mid y - y',$$

czyli że

$$y' \equiv y \pmod{m}.$$

A zatem faktycznie odwrotność x modulo m musi być wyznaczona jednoznacznie.

Pokażmy teraz, że taka odwrotność istnieje. W tym celu udowodnimy, że dla dowolnych liczb całkowitych a i b równanie (zwykłe, nie modulo):

$$au + bv = NWD(a,b) \tag{7}$$

ma zawsze rozwiązanie, w którym u i v są całkowite. Zastosowanie tej własności dla x i m będzie oznaczało, że istnieją liczby całkowite u i v takie, że

$$xu + mv = NWD(x,m) = 1,$$

czyli

$$xu \equiv 1 \pmod{m}.$$

Liczba u będzie więc szukaną odwrotnością x modulo m.

Wróćmy do równania (7). Przypomnijmy, że $NWD(a,b)$ można wyznaczyć za pomocą algorytmu Euklidesa, w którym w każdym kroku redukujemy problem, korzystając z równości:

$$NWD(a,b) = \begin{cases} NWD(b, a \bmod b) & \text{dla } b \neq 0, \\ a & \text{dla } b = 0. \end{cases} \tag{8}$$

Niech

$$(a_1,b_1),\ (a_2,b_2),\ \ldots,\ (a_t,b_t)$$

będą kolejnymi parami liczb, które pojawiają się w trakcie działania algorytmu Euklidesa wywołanego dla liczb a i b. Muszą wówczas zachodzić podstawowe zależności:

$$a_1 = a,\ b_1 = b,\ b_i > 0 \text{ dla } 1 \leqslant i < t,\ b_t = 0$$

oraz

$$NWD(a_1,b_1) = NWD(a_2,b_2) = \ldots = NWD(a_t,b_t) = a_t.$$

Zauważmy, że dla liczb a_t i b_t bardzo łatwo znaleźć rozwiązanie równania (7):

$$a_t u_t + b_t v_t = NWD(a_t,b_t) = a_t, \tag{9}$$

czyli

$$u_t = 1,\ v_t = 0. \tag{10}$$

Z kolei korzystając z rozwiązania u_{i+1}, v_{i+1} dla pary a_{i+1}, b_{i+1}, można stosunkowo prosto wyznaczyć rozwiązanie u_i, v_i dla pary a_i, b_i. Otóż w równaniu

$$a_{i+1}u_{i+1} + b_{i+1}v_{i+1} = NWD(a_{i+1},b_{i+1}) \tag{11}$$

możemy podstawić:

$$NWD(a_{i+1},b_{i+1}) := NWD(a_i,b_i),\ a_{i+1} := b_i \text{ oraz } b_{i+1} := a_i \bmod b_i$$

i otrzymujemy

$$b_i \cdot u_{i+1} + (a_i \bmod b_i) \cdot v_{i+1} = NWD(a_i, b_i). \tag{12}$$

Ponieważ

$$a_i \bmod b_i = a_i - \left\lfloor \frac{a_i}{b_i} \right\rfloor \cdot b_i,$$

to (12) możemy zapisać w postaci

$$b_i \cdot u_{i+1} + \left(a_i - \left\lfloor \frac{a_i}{b_i} \right\rfloor \cdot b_i\right) \cdot v_{i+1} = NWD(a_i, b_i),$$

czyli

$$a_i \cdot v_{i+1} + b_i \cdot \left(u_{i+1} - \left\lfloor \frac{a_i}{b_i} \right\rfloor v_{i+1}\right) = NWD(a_i, b_i).$$

To oznacza, że:

$$u_i = v_{i+1},\ v_i = u_{i+1} - \left\lfloor \frac{a_i}{b_i} \right\rfloor v_{i+1}, \tag{13}$$

są poszukiwanym rozwiązaniem dla a_i oraz b_i. Korzystając z rozwiązania (10) i zależności (13), potrafimy w t krokach znaleźć rozwiązanie dla pary $a = a_1$ i $b = b_1$. To kończy dowód.

Przykład 1. Prześledźmy działanie algorytmu podczas wyznaczania odwrotności 28^{-1} mod 51. W tym celu rozwiążemy równanie

$$28u + 51v = 1,$$

obliczając $NWD(28,51) = 1$ algorytmem Euklidesa. Kolumny a_i oraz b_i poniższej tabeli zawierają argumenty kolejnych wywołań procedury NWD w algorytmie. Kolumny u_i oraz v_i zawierają rozwiązania równań

$$a_i u_i + b_i v_i = 1$$

wyznaczone za pomocą wzorów (10) oraz (13). Strzałki pokazują kolejność wyliczania wartości poszczególnych parametrów.

wywołania rekurencyjne	$\mathbf{a_i}$	$\mathbf{b_i}$	**powroty z wywołań**	$\mathbf{u_i}$	$\mathbf{v_i}$
↓	28	51		−20	11
	51	28	↑	11	−20
	28	23		−9	11
	23	5		2	−9
	5	3		−1	2
	3	2		1	−1
	2	1		0	1
	1	0		1	0

Szukaną odwrotnością 28 modulo 51 jest więc

$$u_1 \bmod b_1 = -20 \bmod 51 = 31.$$

Możemy dodatkowo sprawdzić, że faktycznie:

$$28 \cdot 31 = 868 \equiv 1 \pmod{51}.$$

XX Międzynarodowa Olimpiada Informatyczna,

Kair, Egipt 2008