Marian M. Kędzierski
Treść zadania, Opracowanie

Tomasz Kociumaka, Paweł Parys
Program

Dostępna pamięć: 32 MB.

OI, Etap II, dzień próbny, 08.02.2011

Sejf

Bajtazar jest słynnym kasiarzem, który porzucił przestępczy żywot i zajął się testowaniem zabezpieczeń antywłamaniowych. Właśnie dostał do sprawdzenia nowy rodzaj sejfu: kombinatoryczny. Sejf jest otwierany pokrętłem, które kręci się w kółko. Można je ustawić w n różnych pozycjach ponumerowanych od 0 do $n-1$. Ustawienie pokrętła w niektórych pozycjach powoduje otwarcie sejfu, a w innych nie. Przy tym, pozycje otwierające sejf mają taką własność, że jeżeli x i y są takimi pozycjami, to $(x+y) \bmod n$ też powoduje otwarcie sejfu (dotyczy to także przypadku, gdy $x = y$).

Bajtazar sprawdził k różnych pozycji pokrętła: $m_1, m_2, \ldots, m_k$. Pozycje $m_1, m_2, \ldots, m_{k-1}$ nie powodują otwarcia sejfu, dopiero ustawienie pokrętła w pozycji m_k spowodowało jego otwarcie. Bajtazarowi nie chce się sprawdzać wszystkich pozycji pokrętła. Chciałby wiedzieć, na podstawie do tej pory sprawdzonych pozycji, jaka jest maksymalna możliwa liczba pozycji, w których pokrętło otwiera sejf. Pomóż mu i napisz odpowiedni program.

Wejście

Pierwszy wiersz standardowego wejścia zawiera dwie liczby całkowite n oraz k oddzielone pojedynczym odstępem, $1 \leqslant k \leqslant 250\,000$, $k \leqslant n \leqslant 10^{14}$. W drugim wierszu znajduje się k różnych liczb całkowitych, pooddzielanych pojedynczymi odstępami, $m_1, m_2, \ldots, m_k$, $0 \leqslant m_i < n$. Możesz założyć, że dane wejściowe odpowiadają pewnemu sejfowi spełniającemu warunki zadania.

W testach wartych ok. 70% punktów zachodzi $k \leqslant 1000$. W części tych testów, wartych ok. 20% punktów, zachodzą dodatkowo warunki $n \leqslant 10^8$ oraz $k \leqslant 100$.

Wyjście

Twój program powinien wypisać w pierwszym (i jedynym) wierszu standardowego wyjścia jedną liczbę całkowitą: maksymalną liczbę pozycji pokrętła, które mogą powodować otwarcie sejfu.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
`42 5` `28 31 10 38 24`	`14`

Rozwiązanie

Analiza problemu

Na początku zastanówmy się, jaka może być postać zbioru pozycji otwierających sejf. Z własności sejfu wynika, że dla dowolnej pozycji otwierającej a, pozycje $2a \bmod n$,

$3a \bmod n, \ldots, na \bmod n = 0$, czyli jej wielokrotności modulo n, również wszystkie są otwierające. Przez x oznaczmy pozycję otwierającą o *najmniejszym dodatnim* numerze. Wypiszmy kolejno jej wielokrotności, tzn. $x, 2x, \ldots, nx$. Wiemy, że ich reszty z dzielenia przez n są pozycjami otwierającymi. Zwróćmy szczególną uwagą na tę wielokrotność $w = i \cdot x$, która jako pierwsza osiąga co najmniej n, tzn. $(i-1) \cdot x < n \leqslant i \cdot x$. Wówczas $w \bmod n = w - n$ jest otwierająca, a przy tym $w \in [n, n + x)$, skąd $w \bmod n \in [0, x)$. Jednakże zdefiniowaliśmy x jako *najmniejszą* dodatnią pozycję otwierającą, skąd wnioskujemy, że $w \bmod n = 0$, a więc $n = i \cdot x$. Oznacza to, że liczba x jest *dzielnikiem* n. Dla podkreślenia tego faktu będziemy ją odtąd oznaczać przez d.

Pamiętamy, że wszystkie wielokrotności tej liczby z przedziału $[0, n)$ (jest ich $\frac{n}{d}$) to pozycje otwierające. Okazuje się, że są to jedyne takie pozycje. Weźmy bowiem dowolną pozycję otwierającą a i podzielmy ją z resztą przez d. Mamy $a = q \cdot d + r$ dla pewnych liczb całkowitych q, r, przy czym $0 \leqslant r < d$ oraz $0 \leqslant q < \frac{n}{d}$. Pozycja $(\frac{n}{d} - q) \cdot d$ jest, jak wiemy, także otwierająca, a więc suma obydwu wspomnianych pozycji modulo n, tzn. $((\frac{n}{d} - q) \cdot d + q \cdot d + r) \bmod n = r$, także. Ponieważ $0 \leqslant r < d$, a d była najmniejszą dodatnią pozycją otwierającą, to wnioskujemy, że $r = 0$, a więc a jest wielokrotnością d.

Tym samym zakończyliśmy dowód kluczowej dla zadania obserwacji. Wykazaliśmy, że każdy możliwy zbiór pozycji otwierających to zbiór wielokrotności pewnego dzielnika liczby n. Łatwo sprawdzić, iż każdy zbiór tej postaci spełnia żądaną kombinatoryczną własność sejfu. Aby jednak taki zbiór pozycji otwierających był zgodny z obserwacjami Bajtazara, muszą być spełnione dodatkowo następujące warunki:

- $m_1, m_2, \ldots, m_{k-1}$ nie należą do zbioru pozycji otwierających, a więc nie są wielokrotnościami powyższej liczby d,
- m_k jest pozycją otwierającą, czyli wielokrotnością d.

Szukamy największego zbioru spełniającego wszystkie powyższe warunki, a dla zadanego d taki zbiór liczy $\frac{n}{d}$ elementów. Innymi słowy, chcemy więc znaleźć najmniejszy dzielnik d spełniający podane wyżej warunki.

Na potrzeby dalszych rozważań oznaczmy liczbę wszystkich dzielników liczby n przez $D(n)$.

Rozwiązanie siłowe

Znaleźliśmy proste sformułowanie naszego problemu w języku teorii liczb: należy znaleźć najmniejszy dzielnik d liczby n, który dzieli również m_k, ale nie dzieli żadnej spośród liczb $m_1, m_2, \ldots, m_{k-1}$.

Możemy więc już podać pierwsze rozwiązanie zadania — sprawdzać wszystkie liczby od 1 do n po kolei i dla każdej z nich weryfikować, czy warunki $d \mid n$, $d \mid m_k$, $d \nmid m_1, \ldots, m_{k-1}$ są spełnione. To daje złożoność $O(n \cdot k)$ lub, jeśli dla $d \nmid n$ nie będziemy sprawdzać kolejnych warunków, $O(n + D(n) \cdot k)$. Takie bardzo proste programy otrzymywały 21 punktów, przykładowe implementacje znajdują się w plikach `sejs3.cpp` i `sejs11.pas`.

Pierwsza optymalizacja

Zastanówmy się, gdzie jest pole do przyspieszenia naszego pierwszego algorytmu. Tracimy dużo czasu, niepotrzebnie przeglądając wszystkie liczby od 1 do n — szukamy przecież dzielnika liczby n. Chcielibyśmy zatem przeglądać tylko dzielniki liczby n.

Generowanie dzielników

Wygenerowanie listy wszystkich dzielników danej liczby jest bardzo proste, jeśli satysfakcjonuje nas złożoność $O(\sqrt{n})$. Wystarczy bowiem przeglądać wszystkie liczby od 1 aż do $\lfloor\sqrt{n}\rfloor$ i, jeśli natrafimy na dzielnik d, wstawić na listę również liczbę $\frac{n}{d}$. Jak widać, taki algorytm w każdym przypadku działa w czasie $\Theta(\sqrt{n})$.

Przedstawimy jeszcze inną metodę, także o złożoności pierwiastkowej, która jednak w większości przypadków jest znacznie szybsza, a przy tym pozwoli lepiej zrozumieć strukturę zbioru dzielników. Pierwszą jej fazą jest znalezienie rozkładu na czynniki pierwsze liczby n. Wykorzystamy algorytm działający w czasie $O(\sqrt{n})$, przedstawiony poniżej.

```
1:  ℓ := 0;
2:  i := 2;
3:  while i · i ⩽ n do begin
4:    if n mod i = 0 then begin
5:      { liczba i jest dzielnikiem pierwszym liczby n }
6:      ℓ := ℓ + 1;
7:      p[ℓ] := i;
8:      α[ℓ] := 0;
9:      while n mod i = 0 do begin
10:       α[ℓ] := α[ℓ] + 1;
11:       n := n/i;
12:     end
13:   end
14:   { n jest liczbą pierwszą lub jedynką }
15:   if n > 1 then begin
16:     p[ℓ] := n;
17:     α[ℓ] := 1;
18:     ℓ := ℓ + 1;
19:   end
20: end
```

Pozostawiamy jako proste ćwiczenie obserwację, że $n = p_1^{\alpha_1} \dots p_\ell^{\alpha_\ell}$, gdzie liczby $p_1 < \dots < p_\ell$ są pierwsze, zaś $\alpha_1, \dots, \alpha_n$ dodatnie. Czas działania algorytmu szacuje się przez $O(\log n + \max(p_{\ell-1}, \sqrt{p_\ell}))$. Owa złożoność nigdy nie przekracza $O(\sqrt{n})$, a zazwyczaj, zwłaszcza gdy liczba n ma dużo dzielników (a dla takich działa wolno reszta rozwiązania zadania), jest znacznie mniejsza.

Mając już rozkład na czynniki pierwsze $n = p_1^{\alpha_1} \dots p_\ell^{\alpha_\ell}$, wystarczy zauważyć, że dzielniki liczby n to dokładnie liczby postaci $p_1^{\beta_1} \dots p_\ell^{\beta_\ell}$, gdzie $\beta_i \in \{0, 1, \dots, \alpha_i\}$. Niech kolejnym ćwiczeniem dla Czytelnika będzie skonstruowanie na tej podstawie

algorytmu działającego w czasie $O(D(n))$. Być może prościej na początek pozwolić na czas rzędu $D(n) \cdot \left(\sum_{i=1}^{\ell} \alpha_i\right)$, czyli pesymistycznie $O(D(n) \cdot \log n)$. Warto przy okazji spostrzec, że $D(n) = (\alpha_1 + 1) \ldots (\alpha_\ell + 1)$, co wkrótce będzie przydatne do szacowania $D(n)$.

Otrzymaliśmy algorytm wyznaczania dzielników, który po sfaktoryzowaniu liczby n działa już w optymalnym czasie $O(D(n))$. Łącznie jest to w naszym przypadku wciąż $O(\sqrt{n})$. Poprawienie tej złożoności jest możliwe, ale leży poza wymaganiami stawianymi uczestnikom Olimpiady[1].

Szacowanie liczby dzielników

Dzięki szybszemu generowaniu zbioru dzielników otrzymaliśmy algorytm działający w czasie $O(\sqrt{n} + k \cdot D(n))$. Aby ocenić, czy takie rozwiązanie wystarczy choćby dla $k \leqslant 1000$, przydałoby się nam jakieś oszacowanie na $D(n)$. Oczywiście $D(n) = O(\sqrt{n})$, jednak przy n rzędu 10^{14} takie ograniczenie nas nie satysfakcjonuje. Intuicyjnie wydaje się jasne, że dzielników powinno być znacznie mniej, ale chcielibyśmy wesprzeć swoje podejrzenia konkretami. Zadanie nie pochodzi z I etapu, więc nie chcemy korzystać przy tym z Internetu czy książek. Możemy jednak napisać program, który da nam jakieś konkretne liczbowe ograniczenie[2].

Sposób I

Nie jesteśmy w stanie przejrzeć wszystkich liczb z zakresu $[1, 10^{14}]$, jednak możemy przejrzeć wszystkie liczby z zakresu $[1, 10^7]$. Oznaczmy przez $D(1,\, s)$ maksymalną liczbę dzielników liczby z przedziału $[1, s]$. Wykorzystamy następujący fakt:

Twierdzenie 1. *Dla każdego $s > 1$ zachodzi $D(1,\, s^2) \leqslant 2 \cdot D(1,\, s)^2$.*

Dowód: Rozważmy dowolną liczbę $t \in [1,\, s^2]$ i jej rozkład na czynniki pierwsze

$$t = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_\ell^{\alpha_\ell}.$$

Wiemy, że $D(t) = (\alpha_1+1)\cdot(\alpha_2+1)\cdot\ldots\cdot(\alpha_\ell+1)$. Niech $i_1, i_2, \ldots, i_q$ będą pozycjami, dla których $\alpha_i = 1$, ustawionymi w kolejności rosnącej. Przypiszmy $\beta_{i_{2j-1}} = 1$, $\beta_{i_{2j}} = 0$ dla $j = 1, 2, \ldots$ Dla pozostałych i niech $\beta_i = \lfloor \frac{\alpha_i}{2} \rfloor$, gdzie $\lfloor x \rfloor$ oznacza część całkowitą liczby x. Wówczas liczba

$$u = p_1^{\beta_1} p_2^{\beta_2} \ldots p_\ell^{\beta_\ell}$$

należy do przedziału $[1, s]$. Ponadto $D(t) \leqslant 2 \cdot D(u)^2$, ponieważ

$$(\alpha_{i_{2j-1}} + 1) \cdot (\alpha_{i_{2j}} + 1) = 4 = (\beta_{i_{2j-1}} + 1)^2 \cdot (\beta_{i_{2j}} + 1)^2,$$

$$(\alpha_{i_q} + 1) \leqslant 2 \cdot (\beta_{i_q} + 1)^2,$$

[1] Chodzi tu oczywiście o szybsze metody faktoryzacji, jak choćby opisany w książce [22] niedeterministyczny algorytm „rho" Pollarda. Druga faza wymaga tylko lepszego niż $O(\sqrt{n})$ oszacowania na $O(D(n))$.

[2] Teoretyczne oszacowania asymptotyczne są w tym przypadku i tak mało przydatne. Okazuje się bowiem, że $D(n) = O(n^\varepsilon)$ dla dowolnie małego $\varepsilon > 0$, czyli np. $D(n) = O(n^{0{,}0001})$. Jednak, jak się za chwilę przekonamy, jeszcze dla 14-cyfrowych wartości n zdarza się, że $D(n)$ przekracza $n^{0{,}3}$.

natomiast dla pozostałych i

$$(\alpha_i + 1) \leqslant (\beta_i + 1)^2.$$

Stąd już łatwo wynika teza. ■

Jak można obliczyć $D(1,\, 10^7)$? Możemy, posługując się sitem Eratostenesa, wyznaczyć dla każdej liczby jej *najmniejszy dzielnik pierwszy*, co pozwoli szybko generować rozkłady na czynniki pierwsze. Następnie dla każdej liczby z rozważanego przedziału możemy zbudować jej rozkład, obliczając zarazem liczbę jej dzielników. Taki prosty eksperyment daje nam $D(1,\, 10^7) = 448$. Tak więc $D(1,\, 10^{14}) \leqslant 2 \cdot 448^2 = 401\,408$.

Sposób II

Oszacowanie przez 401 408 nadal jest bardzo zgrubne. Okazuje się jednak, że możemy wyznaczyć $D(1,\, 10^{14})$ dokładnie.

Potrzebne nam tutaj będzie pojęcie *liczby antypierwszej*, które może być Czytelnikowi znane np. z jednego z zadań VIII Olimpiady Informatycznej [8]. Liczba antypierwsza to taka, która ma więcej dzielników niż wszystkie liczby od niej mniejsze. Oczywiście, do obliczenia $D(1,\, s)$ wystarcza znalezienie największej liczby antypierwszej nie większej niż s.

Szukanie liczb antypierwszych opiera się na spostrzeżeniach, że ciąg wykładników $\alpha_1, \alpha_2, \ldots, \alpha_\ell$ w jej rozkładzie na czynniki pierwsze musi być ciągiem nierosnącym (przypominamy, że ciąg p_i był rosnący) oraz że w tym rozkładzie muszą występować kolejne najmniejsze liczby pierwsze, tzn. w przedziale (p_i, p_{i+1}) nie może być innej liczby pierwszej. Te uwagi wystarczają już, żeby znaleźć największą liczbę antypierwszą w przedziale $[1, s]$ przez iterację po wszystkich niemalejących ciągach wykładników odpowiadających liczbom mniejszym niż s. Ponieważ takich ciągów jest stosunkowo niewiele, możemy w taki sposób w krótkim czasie przesiać przedział $[1, 10^{14}]$. Wynik jest następujący: $D(1,\, 10^{14}) = 17\,280$; jest on osiągany przez liczbę antypierwszą 97 821 761 637 600.

Podsumowanie

Wiedząc już, ile w najgorszym przypadku może wynosić $D(n)$, możemy stwierdzić, że programy implementujące powyższe algorytmy o złożoności $O(\sqrt{n} + k \cdot D(n))$ powinny dostawać (i dostawały) przynajmniej 70 punktów. Przykładowe implementacje można znaleźć w plikach `sejs1.cpp`, `sejs10.pas` oraz (z drobnymi optymalizacjami) `sejs2.cpp`, `sejs3.cpp` i `sejs5.cpp`.

Rozwiązanie wzorcowe

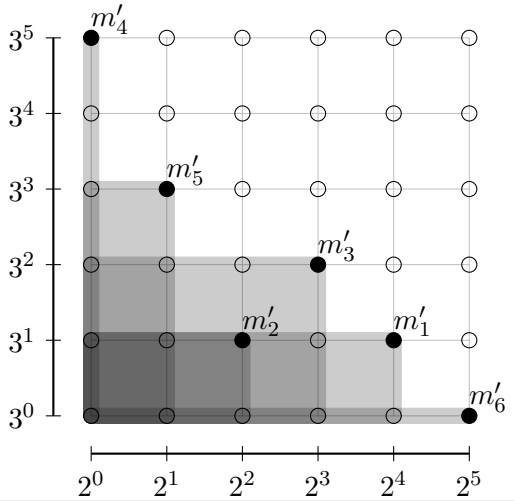
Zanim przejdziemy do kolejnych optymalizacji, dokonamy pewnego uproszczenia, które będzie nas kosztować $O(k \log n)$ (a więc niewiele) czasu, lecz pozwoli rozwiązywać nieco łatwiejszy problem. Otóż, skoro wiemy, że szukane d jest dzielnikiem n, to dla dowolnego i zachodzi równoważność $d \mid m_i \iff d \mid \mathrm{NWD}(m_i, n)$. W szczególności możemy rozważać dzielniki liczby $n' = \mathrm{NWD}(m_k, n)$, a więc zastąpić w sformułowaniu

problemu liczbę n przez n' i pozbyć się warunku związanego z m_k. Ponadto możemy od razu zamienić każdą spośród pozostałych liczb m_i na $m'_i = \mathrm{NWD}(m_i, n')$. W szczególności otrzymujemy algorytm o złożoności $O(\sqrt{n'}+k\cdot(\log n+D(n'))$. Owa złożoność pesymistycznie nie jest jednak lepsza, gdyż m_k może być np. równe 0.

Problemem, jaki nam wciąż pozostaje, jest konieczność wykonywania $O(k)$ kroków algorytmu dla każdego badanego dzielnika liczby n'. Na początek możemy zauważyć, że każde m'_i jest dzielnikiem n', więc wśród nich będzie co najwyżej $D(n') \leqslant 17\,280$ różnych liczb. Tym samym złożoność zmniejsza się do $O(\sqrt{n'}+k\log n+D(n')\cdot\min(k, D(n')))$. Dla nas jest to wciąż zbyt wolno — zajmiemy się konstruowaniem szybszego algorytmu.

Jak więc możemy bardziej zredukować liczbę kroków przeznaczonych na każdy dzielnik? Z pomocą może nam przyjść programowanie dynamiczne, które w przypadku dzielników jest przydatną, lecz nietrywialną metodą. Standardowo programowanie dynamiczne operuje bowiem na tablicach (jedno lub kilkuwymiarowych), rzadziej drzewach czy skierowanych grafach acyklicznych. Widząc zbiór dzielników, niekoniecznie łatwo od razu dostrzec, jak można skonstruować na nim efektywny algorytm dynamiczny.

Zanim podamy rozwiązanie, wprowadźmy pojęcie przydatne do jego opisu. Otóż powiemy, że dzielnik d liczby n' jest *dobry*, jeżeli nie dzieli żadnej spośród liczb $m'_1, \ldots, m'_{k-1}$, w przeciwnym razie będziemy go nazywali *złym*.



Rys. 1: Zbiór dzielników liczby $n' = 6^5 = 7776$ z zaznaczonymi przykładowymi liczbami m'_i dla $i < k$ (pełne kółka). Pozostałe dzielniki są oznaczone pustymi kółkami, złe dzielniki są zamalowane na szaro.

Zachęcamy Czytelnika, aby obejrzawszy rysunek, sam spróbował wymyślić rozwiązanie. Poniżej opisujemy rozwiązanie wzorcowe.

Jak sugeruje rysunek, dwa dzielniki, takie że iloraz jednego przez drugi jest liczbą pierwszą, można traktować jako bliskie sobie, podobnie jak sąsiednie elementy tablicy.

Wówczas złe dzielniki mają regularną strukturę, którą w ogólnym przypadku opisuje następująca obserwacja.

Lemat 1. Niech $d \mid n'$ będzie złym dzielnikiem. Wówczas dla pewnego i zachodzi $d = m'_i$ lub dla pewnej liczby pierwszej p dzielącej n' liczba $d \cdot p$ także jest złym dzielnikiem n'.

Dowód: Jeśli dzielnik d jest zły, to z definicji istnieje takie $i < k$, że $d \mid m'_i$. Jeśli $d = m'_i$, mamy już tezę. W przeciwnym wypadku niech p będzie dowolnym pierwszym dzielnikiem liczby $\frac{m'_i}{d}$ (która oczywiście jest dzielnikiem m'_i, a więc również n'). Wówczas $d \cdot p \mid m'_i$, a więc także jest złym dzielnikiem n'. ■

Wobec tego, jeśli będziemy przeglądać wszystkie dzielniki n' w porządku malejącym, to przy badaniu dzielnika d będziemy już dla wszystkich dzielników postaci $d \cdot p$ wiedzieli, czy są dobre, czy nie. Oczywiście trzeba pamiętać, aby na początku zaznaczyć, że liczby $m'_1, \ldots, m'_{k-1}$ są złe.

Ile zyskujemy w ten sposób? Zamiast $O(\min(k, D(n)))$ kroków algorytmu przeznaczonych na zbadanie jednego dzielnika liczby n' otrzymujemy $O(P(n'))$ kroków, gdzie $P(s)$ to liczba liczb pierwszych dzielących s. Wiemy, że $D(n) \leqslant 17\,280$. Jakie natomiast możemy znaleźć ograniczenie na $P(n)$? Zacznijmy wymnażać kolejne liczby pierwsze, począwszy od 2. Po przemnożeniu pierwszych 13 liczb pierwszych otrzymujemy już wynik większy od 10^{14}. Jeżeli więc $P(s) \geqslant 13$, to nie zwiększając liczby s, możemy zamienić wszystkie wykładniki w jej faktoryzacji na 1, a następnie (potencjalnie) pozmniejszać liczby pierwsze w niej występujące, tak aby były to kolejne liczby pierwsze. Po takich operacjach nadal otrzymamy liczbę większą niż 10^{14}. Wobec tego dla danych z zadania mamy $P(n') \leqslant 12$. Widać, że jest to ponadtysiąckrotnie mniejsze ograniczenie niż poprzednio.

Struktura danych dla dzielników

Pozostaje jeszcze kwestia wyboru struktury danych, w której będziemy przechowywać dzielniki n' wraz z informacją, czy są dobre, złe, czy jeszcze nie zbadane. Poprzednio wystarczała nam zwykła tablica, ponieważ i tak każdy dzielnik rozważaliśmy niezależnie. Teraz potrzebujemy dla danego dzielnika d szybko sprawdzać, czy $d \cdot p$ jest dzielnikiem n, dla kilkunastu różnych liczb pierwszych p. Możemy to robić z użyciem wyszukiwania binarnego w posortowanej tablicy dzielników lub zrównoważonego drzewa poszukiwań binarnych zaimplementowanego za pomocą kontenera `map` z biblioteki STL języka C++. Wtedy każda operacja na tej strukturze będzie kosztowała $O(\log D(n))$. Otrzymamy więc sumaryczną złożoność czasową algorytmu $O(\sqrt{n} + k \log n + D(n)P(n) \log D(n))$, co przy maksymalnych rozmiarach danych daje kilkanaście milionów operacji i wystarczało już do otrzymania 100 punktów.

Istnieje jednak inny, ciekawy pomysł na przechowywanie dzielników n', który nieco jeszcze zmniejsza złożoność asymptotyczną algorytmu. Dosłownie traktuje on analogię między programowaniem dynamicznym na dzielnikach a tym standardowym na tablicy.

Wyobraźmy sobie $P(n')$-wymiarową tablicę *tab*, taką że w komórce $tab[\beta_1][\beta_2]\ldots[\beta_\ell]$ pamiętamy, czy dzielnik

$$p_1^{\beta_1} p_2^{\beta_2} \ldots p_\ell^{\beta_\ell}$$

liczby n' jest dobry, czy zły. Wtedy sprawdzenie dla liczby d liczby $d \cdot p_i$ oznacza po prostu sięgnięcie do sąsiedniej komórki w jednym z wymiarów.

Jak taką tablicę zaimplementować? Stworzenie tablicy o zmiennej liczbie wymiarów może sprawić pewne problemy techniczne (nie radzimy używać 12-wymiarowych tablic czy wektorów), jednak zamiast tworzyć ją jawnie, możemy też symulować ją w tablicy jednowymiarowej *tab1d*. Chcemy zakodować ciągi $\beta_1, \ldots, \beta_\ell$ takie, że $\beta_i \leqslant \alpha_i$ dla każdego i, za pomocą pojedynczych liczb z zakresu $[1, D(n')] = [1, (\alpha_1 + 1) \cdot \ldots \cdot (\alpha_l + 1)]$. Aby to osiągnąć, przypiszmy im numery w kolejności leksykograficznej. Co ciekawe, możemy w ogóle nie konstruować zbioru dzielników, a jedynie wykorzystać następujące odwzorowania:

1. *valueToIndex*(d) zamieniające dzielnik d na indeks odpowiadającego mu pola w tablicy *tab1d*;
2. *indexToValue*(i) dokonujące przeciwnej zamiany;
3. *multiply*(i, j) zwracające liczbę *valueToIndex*$(\mathit{indexToValue}(i) \cdot p_j)$, czyli wartość kolejnego pola w j-tym wymiarze lub informację, że takie pole nie istnieje.

Operacji 1 będziemy potrzebowali $O(k)$ razy (do oznaczenia złych dzielników), operacji 2 — $O(D(n'))$ razy (do sprawdzenia, czy dobry dzielnik jest najmniejszy), zaś operacji 3 — $O(D(n') \cdot P(n'))$ razy. Pierwsze dwie możemy zaimplementować w czasie $O(\log n')$, zaś trzecią w czasie stałym, o ile wcześniej obliczymy pomocniczą tablicę $\mathit{offset}[i] = \Pi_{j=1}^{i}(\alpha_j + 1)$. Dzięki temu otrzymamy algorytm o sumarycznej złożoności

$$O\left(\sqrt{n'} + k \log n + D(n')P(n') + D(n') \log n'\right).$$

Wyniki operacji 2 można także w czasie wygenerować $O(D(n))$ i spamiętać, redukując tym samym złożoność do

$$O\left(\sqrt{n'} + k \log n + D(n')P(n')\right).$$

Szczegóły implementacyjne pozostawiamy jako jeszcze jedno ćwiczenie dla Czytelnika.

Z racji wielu możliwości implementacji poszczególnych faz (dwa sposoby generowania dzielników, kilka struktur danych dla programowania dynamicznego), powstało wiele programów wzorcowych. Znajdują się w plikach `sej.cpp`, `sej[1-8].cpp`, `sej[9-11].pas`. W nagłówku każdego z plików można sprawdzić, których metod w nim użyto.

Heurystyki przyspieszające wolniejsze rozwiązanie

Wróćmy raz jeszcze do rozwiązania, które dla każdego dzielnika liczby n' sprawdzało podzielność wszystkich różnych liczb m'_i przez niego — było to rozwiązanie o złożoności $O(\sqrt{n'} + k \log n + D(n') \cdot \min(k, D(n')))$. Widzimy, że temu rozwiązaniu niewiele brakuje, aby mieścić się w granicach kilku sekund. Okazuje się, że zastosowanie prostej heurystyki dawało w praktyce rozwiązania o czasach działania bliskich rozwiązaniu wzorcowemu, nierzadko nawet niższych. Należy dodać, że podczas zawodów wynik 100

punktów uzyskały tylko takie rozwiązania; wszystkie oparte na pomyśle wzorcowym były, niestety, obarczone drobnymi błędami w implementacji.

Przejdźmy do opisu tejże heurystyki. Oczywiście po znalezieniu takiego m'_i, które jest wielokrotnością badanego dzielnika d liczby n', możemy od razu przerwać badanie d. Przy takim postępowaniu liczyć się zaczyna kolejność, w której sprawdzamy m'_i. Otóż okazuje się, że najbardziej naturalna, rosnąca kolejność wcale nie jest najlepsza, ponieważ małe dzielniki liczb niemal antypierwszych (a dla takich liczb n' algorytm działa powoli) mają mało własnych dzielników, a duże — dużo. Tak więc zamiast rosnącej kolejności liczb m'_i należy rozpatrywać je w kolejności malejącej.

Rozwiązanie takie znajduje się w pliku `sejs9.cpp`. W przypadku wygenerowanych testów to rozwiązanie nieznacznie ustępowało najszybszym spośród rozwiązań wzorcowych, a wyraźnie pokonywało te najwolniejsze[3]. Inne, gorsze rozwiązania oparte na podobnych pomysłach przyspieszenia można znaleźć w plikach `sejs[6-8].cpp`.

Testy

Wśród testów można wyróżnić następujące grupy:

losowe — testy w pełni losowe, oznaczone literką a;

antypierwsze — testy, w których n jest liczbą niemal antypierwszą; zwykle wtedy $D(n)$ jest duże;

antypierwsze szczególne — testy z grupy **antypierwsze**, które dodatkowo mają szczególny warunek na liczbę m_k; testy te są trudne dla powolnych rozwiązań;

specyficzne — testy rozmaitych typów; często n jest w nich liczbą pierwszą (w opisach poniżej $\mathbb{P}$ to zbiór liczb pierwszych, $p_i \in \mathbb{P}$).

Nazwa	n	k	D(n)	Opis
sej1a.in	3 305	10	4	losowy
sej1b.in	840	20	32	antypierwsze
sej1c.in	1	1	1	najmniejszy możliwy test
sej1d.in	1 680	15	40	antypierwsze, $m_k = 0$
sej2a.in	566 345	50	8	losowy
sej2b.in	1 081 080	80	256	antypierwsze
sej2d.in	665 280	53	224	antypierwsze, $m_k = 0$

[3]Wiadomo, że heurystyka ta nieco poprawia złożoność, ale nie tłumaczy to w pełni jej znakomitego działania nawet dla n wyraźnie większych niż 10^{14}. Pozostaje ciekawe pytanie, czy istnieją takie testy, w których spisywałaby się istotnie gorzej niż na tych, gdzie m'_i to wszystkie dzielniki n' mniejsze niż ustalona wartość. Tak właśnie, z bardzo drobnymi losowymi modyfikacjami, wyglądała podczas zawodów większość testów wydajnościowych.

Nazwa	n	k	D(n)	Opis
sej3a.in	88 565 049	100	8	losowy
sej3b.in	73 513 440	100	768	antypierwsze
sej3d.in	73 513 440	89	768	antypierwsze, $m_k = 0$
sej4a.in	7 300 943 302	200	4	losowy
sej4b.in	10 475 665 200	250	2 400	antypierwsze
sej4d.in	6 983 776 800	342	2 304	antypierwsze, $m_k = 0$
sej5a.in	945 494 546 981	500	6	losowy
sej5b.in	963 761 198 400	600	6 720	antypierwsze
sej5c.in	999 999 999 989	2	2	$n \in \mathbb{P}$, $m_k = 0$
sej5d.in	963 761 198 400	656	6 720	antypierwsze, $m_k = 0$
sej6a.in	17 536 146 808 269	1 000	32	losowy
sej6b.in	55 898 149 507 200	1 000	15 360	antypierwsze
sej6c.in	99 999 999 999 973	1	2	$n \in \mathbb{P}$, $m_1 \neq 0$
sej6d.in	55 898 149 507 200	970	15 360	antypierwsze, $m_k = 0$
sej7a.in	5 460 205 985 383	1 000	4	losowy
sej7b.in	65 214 507 758 400	1 000	16 128	antypierwsze
sej7c.in	99 989 999 999 941	1 000	2	$n \in \mathbb{P}$, $m_k = 0$
sej7d.in	32 607 253 879 200	960	13 824	antypierwsze, $m_k = 0$
sej8a.in	59 161 962 579 511	1 000	4	losowy
sej8b.in	93 163 582 512 000	1 000	16 384	antypierwsze
sej8c.in	99 999 999 999 962	1 000	4	$n = 2p_1$, $d = 2$
sej8d.in	55 898 149 507 200	992	15 360	antypierwsze, $m_k = 0$
sej9a.in	68 039 674 035 556	1 000	12	losowy
sej9b.in	97 821 761 637 600	1 000	17 280	antypierwsze
sej9c.in	99 998 289 000 187	1 000	4	$n = p_2 \cdot p_3$
sej9d.in	93 163 582 512 000	978	16 384	antypierwsze, $m_k = 0$
sej10a.in	50 807 342 460 355	1 000	64	losowy
sej10b.in	100 000 000 000 000	1 000	225	antypierwsze
sej10c.in	97 999 999 999 994	1 000	4	$n = 2p_4$, sejf otwierają: 0 i $\frac{n}{2}$
sej10d.in	93 163 582 512 000	1 000	16 384	antypierwsze, $m_k = 0$
sej11a.in	69 909 258 653 683	100 000	4	losowy
sej11b.in	97 821 761 637 600	100 000	17 280	antypierwsze
sej11c.in	97 821 761 637 600	100 000	17 280	antypierwsze, duże m_k

Nazwa	n	k	D(n)	Opis
sej11d.in	97 821 761 637 600	149 000	17 280	antypierwsze, $m_k = 0$
sej11e.in	97 821 761 637 600	100 500	17 280	antypierwsze, $m_k = 0$
sej12a.in	31 451 726 809 867	200 000	32	losowy
sej12b.in	97 821 761 637 600	200 000	17 280	antypierwsze
sej12c.in	48 910 880 818 800	196 054	14 400	antypierwsze, duże m_k
sej12d.in	65 214 507 758 400	156 200	16 128	antypierwsze, $m_k = 0$
sej12e.in	97 821 761 637 600	196 534	17 280	antypierwsze, $m_k = 0$
sej13a.in	16 016 997 334 119	250 000	8	losowy
sej13b.in	97 821 761 637 600	250 000	17 280	antypierwsze
sej13c.in	97 821 761 637 600	241 542	17 280	antypierwsze, duże m_k
sej13d.in	97 821 761 637 600	245 434	17 280	antypierwsze, $m_k = 0$
sej13e.in	97 821 761 637 600	249 542	17 280	antypierwsze, $m_k = 0$
sej14a.in	87 609 341 358 200	250 000	48	losowy
sej14b.in	97 821 761 637 600	250 000	17 280	antypierwsze
sej14c.in	93 163 582 512 000	250 000	16 384	antypierwsze, duże m_k
sej14d.in	97 821 761 637 600	250 000	17 280	antypierwsze, $m_k = 0$
sej14e.in	97 821 761 637 600	249 050	17 280	antypierwsze, $m_k = 0$