Informatyczny kącik olimpijski (28): Maszyna Fibonacciego

W tym kąciku zajmiemy się zadaniem z finału Potyczek Algorytmicznych 2009.

Weźmy funkcję F zwracającą liczby Fibonacciego, tzn. $F(0) = 0$, $F(1) = 1$ oraz $F(m) = F(m-1) + F(m-2)$ dla $m \geqslant 2$. Mamy ciąg rejestrów $(i_1, i_2, \ldots, i_n)$, początkowo ustawionych na zera. W zadaniu chodzi o zaimplementowanie dwóch operacji:

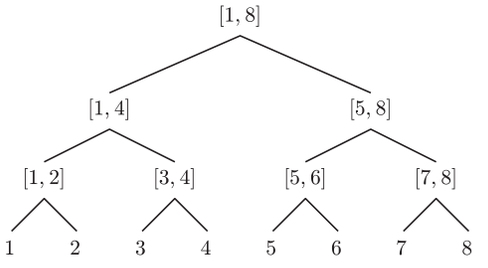
- dla podanych a i b dodanie jedynki do każdego z rejestrów $i_a, i_{a+1}, \ldots, i_{b-1}, i_b$,
- dla podanych a i b wypisanie reszty z dzielenia wartości $F(i_a) + F(i_{a+1}) + \ldots + F(i_{b-1}) + F(i_b)$ przez $P = 10^9 + 7$.

Wykonamy łącznie k operacji tych dwóch typów.

Zacznijmy od jakiegokolwiek poprawnego rozwiązania. Naturalnie, łatwo w czasie $O(n)$ wykonać pierwszą operację poprzez dodanie odpowiednich jedynek. Druga operacja zajmuje w takim razie czas $O(i_a + i_{a+1} + \ldots + i_b)$, gdyż $F(x) \bmod P$ obliczamy łatwo w czasie $O(x)$, a ponieważ $i_y \leqslant k$, więc daje to złożoność $O(n \cdot k)$. Łączny koszt takiego algorytmu to $O(n \cdot k^2)$.

Mądrzej jednak będzie pamiętać od razu wartości $F(i_y) \bmod P$ zamiast samych i_y. Aby móc zwiększać i_y o jeden, potrzebne nam są tak naprawdę dwie kolejne wartości F, np. $A_y = F(i_y) \bmod P$ oraz $B_y = F(i_y + 1) \bmod P$. Zaczniemy więc od $A_y = 0$, $B_y = 1$ dla $y = 1, 2, \ldots, n$. W ten sposób zarówno pierwszą (podstawiając, dla $a \leqslant y \leqslant b$, $(A_y, B_y) = (B_y, (A_y + B_y) \bmod P)$), jak i drugą (zwracając $(A_a + \ldots + A_b) \bmod P$) operację wykonujemy w czasie $O(n)$ i dostajemy sumaryczną złożoność $O(n \cdot k)$.

Aby jeszcze przyspieszyć rozwiązanie, kluczowe jest spostrzeżenie, że przekształcenie $(x, y) \to (y, x + y)$ jest liniowe, a więc ma swoją macierz L (rozmiaru 2×2), tzn. taką macierz, że $L \cdot (x, y) = (y, x + y)$. Ponadto, k-krotne złożenie takiego przekształcenia też jest liniowe i ma macierz L^k (oczywiście też rozmiaru 2×2). Dodatkowo, możemy bezkarnie przyjąć, że $n = 2^m$ dla m naturalnego. W przeciwnym bowiem razie zwiększamy n do najbliższej potęgi dwójki, co nie zmienia złożoności (bo n wzrasta co najwyżej dwukrotnie), i później po prostu nie używamy w ogóle części rejestrów. Wyobraźmy sobie teraz pełne drzewo binarne o $2^{m+1} - 1$ wierzchołkach (patrz rysunek), w którego liściach mamy wartości A_y i B_y dla kolejnych y z przedziału $[1, 2^m]$.



W każdym węźle v zapamiętamy informację o:

- synach tego wierzchołka w drzewie: $lewy_v$ i $prawy_v$ (dla liści nieokreślone),
- przedziale rejestrów będących jego potomkami w drzewie, tj. $[l_v, p_v]$,
- sumach wartości A_y i B_y po tych rejestrach, odpowiednio: A'_v i B'_v,
- macierzy przekształcenia L_v, które zostało wykonane na całym przedziale rejestrów $[l_v, p_v]$.

Początkowo, dla wszystkich v, L_v jest macierzą identyczności. W momencie dodawania jedynki do rejestrów z przedziału $[a, b]$, zmieniamy A'_v, B'_v i L_v dla pewnych wierzchołków, a konkretnie dla takich, żeby ich przedziały potomków sumowały się w sposób rozłączny (biorąc pod uwagę tylko zawarte w przedziale liczby całkowite) do przedziału $[a, b]$, np. $[2, 7] = [2, 2] \cup [3, 4] \cup [5, 6] \cup [7, 7]$. Tych przedziałów wybierzemy $O(m)$, wywołując PODZIEL$(a, b,$ korzeń$)$ – w poniższym pseudokodzie zakładamy wykonywanie działań modulo P:

```
PRZYŁÓŻ(v, X)
(A'_v, B'_v) := X · (A'_v, B'_v)
L_v := L_v · X

PODZIEL(a, b, v)
 1  if (a = l_v) and (b = p_v) then
 2    PRZYŁÓŻ(v, L)
 3  else
 4    PRZYŁÓŻ(lewy_v, L_v)
 5    PRZYŁÓŻ(prawy_v, L_v)
 6    L_v := id_2
 7    if a ⩽ p_{lewy_v} then
 8      PODZIEL(a, min(b, p_{lewy_v}), lewy_v)
 9    if b ⩾ l_{prawy_v} then
10      PODZIEL(max(a, l_{prawy_v}), b, prawy_v)
11    A'_v := A'_{lewy_v} + A'_{prawy_v}
12    B'_v := B'_{lewy_v} + B'_{prawy_v}
```

Dlaczego to działa? Po pierwsze, kończy się, bo w każdym wywołaniu PODZIEL zachodzi: $[a, b] \subset [l_v, p_v]$, a dla liści mamy $l_v = p_v$, więc zachodzi warunek z linii pierwszej i funkcja nie wywołuje się więcej rekurencyjnie. Po drugie, rzeczywiście rozkłada przedział $[a, b]$ na sumę przedziałów, ponieważ $p_{lewy_v} + 1 = l_{prawy_v}$, więc przedziały $[a, \min(b, p_{lewy_v})]$ i $[\max(a, l_{prawy_v}), b]$ (lub jeden z nich, gdy nie zachodzi któryś z warunków z linii 7 i 9) pokrywają cały przedział $[a, b]$. Po trzecie wreszcie, po pierwszym takim wywołaniu, które powoduje rozgałęzienie rekurencyjne (tzn. wykonują się obie linie 8 i 10), w każdym kolejnym zachodzi co najmniej jeden z warunków: $a = l_v$ lub $b = p_v$. W takim razie każde kolejne rozgałęzienie rekurencyjne powoduje, że w co najmniej jednym z dwóch wywołań rekurencyjnych zachodzi zarówno $a = l_v$, jak i $b = p_v$, a więc ta gałąź natychmiast się kończy. Stąd, wywołań funkcji PODZIEL może być co najwyżej $4m + 1$ (jedno w korzeniu oraz w każdym z dwóch poddrzew po $2m$: m takich, które od razu się kończą, i m kontynuowanych), a więc $O(m)$. To kończy uzasadnienie, że potrafimy za pomocą takiej struktury w czasie $O(\log n)$ wykonać pierwszy typ operacji. Typ drugi obsługujemy analogicznie, z tą różnicą, że odpowiednia funkcja PODZIEL2 zwraca żądaną sumę. W tym celu linia druga zostaje zmieniona na **return** A'_v, a suma wyników z podwywołań PODZIEL2 z linii 8 i 10 zostaje zwrócona w dodatkowej linii 13 jako wynik wywołania tejże funkcji. Dowód poprawności i złożoności czasowej jest analogiczny. Stąd łączny czas wykonania wynosi $O(n + k \log n)$ przy zużyciu pamięci rzędu $O(n)$, gdyż drzewo ma co najwyżej $4n$ wierzchołków i w każdym przechowujemy stałą ilość informacji.

Tomasz KULCZYŃSKI