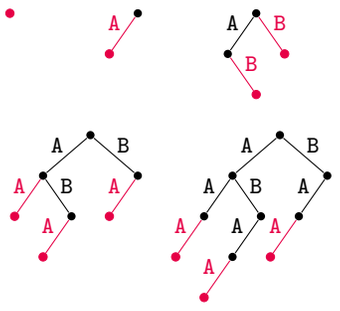
Zliczamy podciągi

*Tomasz IDZIASZEK**

* Informatyk, prowadzi stronę internetową `algonotes.com`

Zaczniemy od takiego zadania: dla danego n-literowego słowa s chcemy znaleźć liczbę jego różnych podciągów. Innymi słowy, chcemy odpowiedzieć na pytanie, ile różnych słów możemy uzyskać poprzez wykreślanie niektórych liter ze słowa s. Dla przykładu rozważmy słowo `ABAA`. Ma ono dokładnie 10 różnych podciągów: słowo puste, `A`, `B`, `AA`, `AB`, `BA`, `AAA`, `ABA`, `BAA` oraz `ABAA`. Dla uproszczenia będziemy rozważać słowa złożone z liter `A` i `B`, ale nasze rozwiązania będą działać dla dowolnego A-literowego alfabetu.



Rys. 1. Konstrukcja drzewa trie dla kolejnych prefiksów słowa `ABAA`. Krawędzie dodawane w kolejnych krokach zaznaczono kolorem; wynikowe drzewo ma $w = 10$ węzłów

Zaczniemy od algorytmu, który będzie konstruował możliwe do uzyskania podciągi dla kolejnych prefiksów słowa s (rys. 1). Wygodnie jest trzymać te podciągi na drzewie, w którym krawędzie są etykietowane literami, a każda ścieżka od korzenia do dowolnego węzła odpowiada jednemu podciągowi (czyli na tzw. drzewie trie). Tak więc liczba węzłów w tego drzewa będzie oznaczała liczbę różnych podciągów (włączając korzeń drzewa odpowiadający podciągowi pustemu).

Przypuśćmy, że skonstruowaliśmy drzewo dla prefiksu $p = s_1 s_2 \dots s_{i-1}$ i chcemy dodać kolejną literę $c = s_i$, aby uzyskać drzewo dla prefiksu $pc = s_1 s_2 \dots s_{i-1} s_i$. Wszystkie podciągi z pc będą albo podciągami występującymi już w p, albo tymi samymi podciągami rozszerzonymi o literę c. Tak więc, gdy w drzewie dla p do każdego węzła dodamy krawędź o etykiecie c, uzyskamy drzewo dla pc. Może się zdarzyć, że w niektórych węzłach taka krawędź już istniała – oznacza to, że odpowiadający podciąg już występował w p, zatem nie należy go dodawać ponownie.

Taki algorytm, choć poprawny, ma złożoność wykładniczą względem n. Istotnie, słowo złożone z n różnych liter ma 2^n podciągów. Ale dla dwuliterowego alfabetu nie jest dużo lepiej: słowo $(\texttt{AB})^{n/2}$ (`AB` powtórzone $\frac{n}{2}$ razy) ma więcej niż $2^{n/2}$ podciągów (w szczególności zawiera wszystkie możliwe $\frac{n}{2}$-literowe słowa jako podciągi).

Okazuje się, że aby znaleźć liczbę podciągów, nie musimy trzymać w pamięci całego drzewa. Niech w_c oznacza liczbę węzłów drzewa, z których wychodzi krawędź z literą c (jest to też po prostu liczba krawędzi z etykietą c). Gdy dodajemy literę c, dodajemy nową krawędź do dokładnie $w - w_c$ węzłów, a zatem zwiększamy sumaryczną liczbę węzłów (tym samym liczbę podciągów) z w na $2w - w_c$. Ponadto zwiększamy liczbę krawędzi z etykietą c z w_c do $w_c + (w - w_c) = w$.

Wystarczy więc, że będziemy trzymali w pamięci wektor $(w, w_{\texttt{A}}, w_{\texttt{B}})$, początkowo równy $(1, 0, 0)$. Gdy dodajemy nową literę `A`, to zastępujemy ten wektor przez $(2w - w_{\texttt{A}}, w, w_{\texttt{B}})$, a dla litery `B` przez wektor $(2w - w_{\texttt{B}}, w_{\texttt{A}}, w)$. Ponieważ zawsze zmieniamy tylko dwie współrzędne wektora, to dostajemy rozwiązanie o złożoności czasowej $O(n + A)$.

W konkursach programistycznych panuje moda na utrudnianie zadań z ciągami przez rozważanie wielu zapytań o fragmenty słowa. Spróbujmy zmierzyć się z taką wersją zadania. Dostajemy zatem q zapytań, każde postaci (l, r), o liczbę różnych podciągów dla fragmentu $s_l s_{l+1} \dots s_r$. Przy czym q jest duże, więc nie możemy sobie pozwolić na uruchomienie algorytmu liniowego dla każdego zapytania oddzielnie.

W algorytmie dla jednego zapytania utrzymujemy wektor $(w, w_{\texttt{A}}, w_{\texttt{B}})$. Ponieważ współczynniki nowego wektora (po dodaniu litery) są kombinacjami liniowymi współczynników oryginalnego wektora, więc możemy zamianę wektora zastąpić mnożeniem go z prawej strony przez jedną z poniższych macierzy:

$$M_{\texttt{A}} = \begin{pmatrix} 2 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad M_{\texttt{B}} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}.$$

Nadużywając nieco notacji, oznaczmy przez M_i macierz odpowiadającą i-tej literze słowa s, czyli $M_i = M_{s_i}$. Zaczynając od wektora $(1, 0, 0)$, mnożymy go przez kolejne wartości M_i, uzyskując na końcu wektor $(w, w_{\texttt{A}}, w_{\texttt{B}})$ dla całego słowa. Pomnożywszy go skalarnie przez $(1, 0, 0)$, dostajemy wartość w, czyli szukaną

Rozwiązanie zadania F 1003.
Spadek z wysokości h_0 trwa $t_0 = \sqrt{2h_0/g}$, po czym następuje odbicie z prędkością $v_1 = k\sqrt{2gh_0}$, a następnie ruch w górę do wysokości $h_1 = k^2 h_0$ (po odbiciu energia kinetyczna wynosi k^2 energii przed odbiciem) i ponowne spadanie. Całkowity czas pomiędzy pierwszym i drugim odbiciem wynosi więc $t_1 = 2\sqrt{2h_1/g} = 2k\sqrt{2h_0/g}$. Analogicznie czas pomiędzy odbiciem n i $n+1$ wynosi:

$$t_n = kt_{n-1} = 2k^n \sqrt{2h_0/g}.$$

Całkowity czas t, jaki upłynie do wytłumienia podskoków, równy jest:

$$t = t_0 + \sum_{n=1}^{\infty} t_n = \sqrt{2h_0/g}\left(1 + \frac{2k}{1-k}\right).$$

W obliczeniach skorzystaliśmy ze wzoru na sumę szeregu geometrycznego. Po podstawieniu danych liczbowych otrzymujemy $t \approx 2{,}6$ s. Model Newtona jest nieco uproszczony i nie uwzględnia wzrastania współczynnika restytucji od 0,7 do 1 wraz ze zmniejszaniem prędkości zderzających się ciał.

liczbę podciągów. Ponieważ macierze są rozmiarów $(A+1)\times(A+1)$ i umiemy pomnożyć dwie takie macierze w czasie $O(A^3)$, to cały algorytm, sprowadzający się do obliczenia wzoru

$$(1,0,0)\ M_1M_2\cdots M_{n-1}M_n\ (1,0,0)^T,$$

zadziała w czasie $O(nA^3)$. Jest to istotnie gorsza złożoność, niż mieliśmy wcześniej, ale przedstawienie obliczeń w postaci macierzowej daje nam większą elastyczność. Obliczenie odpowiedzi dla fragmentu $s_ls_{l+1}\ldots s_r$ wymaga bowiem przemnożenia macierzy $(1,0,0)\ M_lM_{l+1}\cdots M_{r-1}M_r\ (1,0,0)^T$. Ponieważ mnożenie macierzy jest operacją łączną, więc w tym celu możemy użyć struktury danych zwanej drzewem przedziałowym. W liściach drzewa będziemy trzymać macierze $M_1, M_2, \ldots, M_n$, a w węzłach wewnętrznych przemnożone macierze z dzieci. Dzięki temu będziemy mogli odpytywać o iloczyn macierzy dla dowolnego fragmentu w czasie $O(A^3\log n)$, gdyż dzielimy go na $O(\log n)$ przedziałów bazowych, których macierze mnożymy w czasie $O(A^3)$. Z kolei sama konstrukcja drzewa zajmie czas $O(nA^3)$, więc algorytm będzie działał w sumarycznej złożoności czasowej $O(nA^3+qA^3\log n)$.

Konstrukcję drzewa przedziałowego przechowującego macierze opisywaliśmy m.in. w informatycznym kąciku olimpijskim w Δ^3_{10}.

Można ją jeszcze trochę przyspieszyć, korzystając ze standardowej sztuczki dla zapytań o iloczyny macierzy. Tak naprawdę nie pytamy się o całą macierz, a o jeden z jej elementów (dlatego mnożymy obustronnie przez wektor). Ponieważ mnożenie macierzy przez wektor działa w czasie $O(A^2)$, jest więc szybsze niż mnożenie macierzy przez macierz (i daje w wyniku wektor), możemy zatem macierze dla przedziałów bazowych od razu domnażać do jednego z tych wektorów. Tym sposobem algorytm będzie działał w czasie $O(nA^3+qA^2\log n)$.

Macierz M jest odwracalna, jeśli istnieje taka macierz M^{-1}, że iloczyny $M\cdot M^{-1}$ oraz $M^{-1}\cdot M$ są równe macierzy identycznościowej.
Dla odwracalnych macierzy M_1 i M_2 mamy $(M_1\cdot M_2)^{-1}=M_2^{-1}\cdot M_1^{-1}$.

Własność mnożenia macierzy, którą wykorzystujemy w drzewie przedziałowym, to łączność. Gdyby dodatkowo nasze macierze $M_\mathtt{A}$ i $M_\mathtt{B}$ były odwracalne, to zamiast drzewa przedziałowego moglibyśmy wykorzystać zwykłe sumy (a właściwie iloczyny) prefiksowe. Jeśli przyjmiemy oznaczenie $P_i=M_1M_2\cdots M_{i-1}M_i$ oraz $Q_i=P_i^{-1}$, to wtedy $Q_i=M_i^{-1}M_{i-1}^{-1}\cdots M_2^{-1}M_1^{-1}$ oraz

$$(1,0,0)\ M_lM_{l+1}\cdots M_{r-1}M_r\ (1,0,0)^T=(1,0,0)\ Q_{l-1}\cdot P_r\ (1,0,0)^T.$$

Niestety, nie wszystkie macierze są odwracalne. Ale popatrzmy na macierz $M_\mathtt{A}$ jako przekształcającą wektor $v=(w,w_\mathtt{A},w_\mathtt{B})$ na wektor $v'=(w',w'_\mathtt{A},w'_\mathtt{B})$, gdzie $w'=2w-w_\mathtt{A}$, $w'_\mathtt{A}=w$ i $w'_\mathtt{B}=w_\mathtt{B}$. Gdybyśmy dostali wektor v', to czy umielibyśmy na jego podstawie odtworzyć wektor v? Odpowiedź jest twierdząca – proste przekształcenia prowadzą do wzoru $w=w'_\mathtt{A}$, $w_\mathtt{A}=2w'_\mathtt{A}-w'$ i $w_\mathtt{B}=w'_\mathtt{B}$. Są to również przekształcenia liniowe, więc możemy je zapisać w formie macierzy, która musi być zatem macierzą odwrotną do $M_\mathtt{A}$ (tak samo odwracamy macierz $M_\mathtt{B}$):

$$M_\mathtt{A}^{-1}=\begin{pmatrix}0&-1&0\\1&2&0\\0&0&1\end{pmatrix},\qquad M_\mathtt{B}^{-1}=\begin{pmatrix}0&0&-1\\0&1&0\\1&0&2\end{pmatrix}.$$

Możemy więc w czasie $O(nA^3)$ wyznaczyć wszystkie P_i, biorąc $P_i=P_{i-1}M_i$. Macierz odwrotną $Q_i=P_i^{-1}$ możemy również obliczyć w czasie $O(A^3)$, ale jest to bardziej kłopotliwe niż mnożenie. Aby tego uniknąć, wyznaczymy Q_i, korzystając ze wzoru

$$Q_i=(P_{i-1}M_i)^{-1}=M_i^{-1}P_{i-1}^{-1}=M_i^{-1}Q_{i-1}.$$

Teraz jedno zapytanie będzie działać w czasie $O(A^3)$ lub nawet w czasie $O(A^2)$, bo dla wyniku potrzebujemy wykonać mnożenie $(1,0,0)\ Q_{l-1}$ oraz mnożenie $P_r\ (1,0,0)^T$, a następnie pomnożyć skalarnie uzyskane wektory. Ale jeśli przyjrzymy się temu wzorowi bliżej, to tak naprawdę mnożymy w nim pierwszy wiersz macierzy Q_{l-1} przez pierwszą kolumnę macierzy P_r, zatem możemy to bezpośrednio zrobić w czasie $O(A)$. Dostajemy zatem algorytm o złożoności czasowej $O(nA^3+qA)$.

Przypomnijmy, że zaczęliśmy od rekurencji liniowej, którą zapisaliśmy w postaci macierzy, aby skorzystać z ich własności (łączności dla drzewa przedziałowego i istnienia odwrotności dla iloczynów prefiksowych). Zauważmy, że o ile w drzewie przedziałowym potrzebowaliśmy umieć mnożyć dowolne macierze, to w algorytmie sum prefiksowych wystarczy nam domnażanie przez macierze M_i oraz M_i^{-1}, więc znowu możemy skorzystać z faktu, są one szczególnej postaci. Domnażanie macierzy P przez M_i modyfikuje jedynie dwie kolumny macierzy P (przykładowo dla $M_\mathtt{A}$ na drugą kolumnę kopiujemy pierwszą, a pierwszą mnożymy przez 2 i odejmujemy drugą). Zatem możemy skopiować macierz P_{i-1} do macierzy P_i w czasie $O(A^2)$, a następnie zrobić uaktualnienie dwóch kolumn w czasie $O(A)$. (Analogicznie dla macierzy Q.) Tak więc fazę obliczeń wstępnych możemy zrealizować w czasie $O(nA^2)$, co da nam algorytm o złożoności $O(nA^2+qA)$.

Czas na ostatnią obserwację: wcale nie musimy pracowicie kopiować całych macierzy. Ponieważ z macierzy Q_i potrzebujemy jedynie pierwszego wiersza, a z macierzy P_i jedynie pierwszej kolumny, zatem wystarczy te macierze modyfikować w miejscu (czyli nadpisując nieaktualne wartości nowymi w tym samym miejscu), a kopiować jedynie potrzebne wiersze i kolumny, co zajmie czas $O(A)$. Zatem ostatecznie dostajemy rozwiązanie o złożoności czasowej $O(nA+qA)$.