Ezoteryczne Kartki
Niestandardowe hashe
- # -

Jakub Bachurski

wersja 1.4.0

Wstęp

Hashowanie to proces, w którym zastępujemy pewne skomplikowane, pamięciożerne, trudne do porównywania i przechowywania dane ich uproszczoną wersją. Tym uproszczeniem jest najczęściej liczba, która w naturalny sposób jest przechowywana w pamięci komputera. Naturalną konsekwencją takiej czynności jest możliwość powtórzenia się dwóch hashy – taką sytuację nazywamy *kolizją* hashy. Zatem naszym głównym zadaniem będzie opracowanie takich metod hashowania, aby zredukować szansę kolizji, a z drugiej strony pozwolić nam na efektywne wykonywanie wymaganych operacji.[1]

Rozważymy różne przykłady hashowania, od podstawowych hashy wielomianowych, pozwalających na hashowanie ciągów, po różnorodne wersje hashowania zbiorów.

1 Obserwacje

- Opłaca się, aby hashe były liczbami, bo najłatwiej je przechowywać i wykonywać na nich operacje.
- Arytmetyka modulo jest przydatna, ze względu na to, że i tak musimy mieć skończoną liczbę możliwych hashy (liczby dowolnej precyzji odpadają). Przy okazji w charakterze modulo mogą wystąpić liczby pierwsze, które mają przydatne własności związane z losowością.
- Dla pewnej funkcji hashującej $\mathcal{H}$ i obiektów a, b mamy:

  $$a = b \Rightarrow \mathcal{H}(a) = \mathcal{H}(b)$$

  Czyli możemy policzyć hashe dwóch obiektów, i jeżeli były one takie same, to w szczególności ich hashe są równe.

[1] Autor jest zdania, że nazwa „hash” jest ładna i nie należy jest zastępować spolszczonym „hasz”, ani tłumaczeniem „skrót”.

2 Ciągi

Chcemy opracować wygodny algorytm pozwalający na hashowanie ciągu $a = (a_1, a_2, ..., a_n)$ i zastąpienie go liczbą – oznaczmy ją $\mathcal{H}(a)$. Zależy nam, aby hash był prosty do obliczenia i w miarę losowy, aby zredukować szansę kolizji (małe zmiany w ciągu powinny skutkować dużymi zmianami w hashu).

Często zajmując się ciągami (głównie w algorytmach tekstowych) spotkamy się z następującym zgeneralizowanym problemem: wyszukiwaniem wzorca w tekście. Poszukując wzorca a w tekście t i wykorzystując pewien hash $\mathcal{H}$, wystarczy przejrzeć wszystkie podsłowa t wielkości $|a|$, i sprawdzić, czy hash któregoś z nich jest równy $\mathcal{H}(a)$. Taka metoda nosi nazwę algorytmu Rabina-Karpa. Zatem zależy nam, aby $\mathcal{H}$ było łatwe do policzenia dla każdego z podsłów t.

Aby uprościć nasze rozważania, przyjmijmy, że a_i są liczbami całkowitymi oraz $1 \leqslant a_i \leqslant \sigma$. Wprowadźmy też oznaczenie $a[i:j] = (a_i, a_{i+1}, ..., a_{j-1}, a_j)$.

2.1 # Wielomianowy (Rolling hash)

Najczęściej wykorzystywanym hashem jest hash wielomianowy, i to od niego zaczniemy. Właśnie on w wygodny sposób pozwala nam na łatwe rozwiązywanie różnorodnych problemów wyszukiwania wzorca w tekście.

Pomysł jest bardzo prosty. Skorzystamy z zapowiadanej arytmetyki modulo p i wyznaczmy pewną stałą $c > \sigma$ (podstawę). Zacznijmy od $h_0 = 0$ i przejrzyjmy po kolei $x = a_1, a_2, ..., a_n$ wykonując następujący proces dla $i = 1, 2, ..., n$:

$$h_i := ch_{i-1} + a_i \mod p$$

Tak otrzymane h_n jest równe $\mathcal{H}(a)$. Zauważmy, że ponieważ $c > \sigma$, znalezienie kontrprzykładu jest nietrywialne. Ten prosty pomysł możemy rozpisać w postaci, która przypomina obiecany wielomian:

$$\mathcal{H}(a) = c^{n-1}a_1 + c^{n-2}a_2 + ... + c^1 a_{n-1} + c^0 a_n \mod p$$

$$\mathcal{H}(a) = \sum_{i=1}^{n} c^{n-i} a_i \mod p$$

Kluczową własnością jest fakt, że łatwo możemy policzyć hash podsłowa, czyli $\mathcal{H}(a[i:j])$. Jak? Okazuje się, że aby to zrobić, wystarczy spamiętać prefiksowe hashe h_i. Spróbujmy zapisać $\mathcal{H}(a[i:j])$, wykorzystując h:

$$\mathcal{H}(a[i:j]) = \sum_{k=i}^{j} c^{j-k} a_k = \underbrace{\left(\sum_{k=1}^{j} c^{j-k} a_k\right)}_{h_j} - \underbrace{\left(\sum_{k=1}^{i-1} c^{j-k} a_k\right)}_{c^{j-(i-1)} h_{i-1}} \pmod p$$

$$\mathcal{H}(a[i:j]) = h_j - c^{j-i+1} h_{i-1} \pmod p$$

Ten jakże zadowalający wniosek pozwala nam na prostą implementację tego hashowania. W wyprowadzeniu posłużyliśmy się spróbowaliśmy „upodobnić" sytuację do sum prefiksowych, i okazało się, że było to opłacalne.

2.2 Implementacja

Całość implementujemy w $O(n)$, spamiętując hashe prefiksowe h i potęgi c. Osobiście indeksuję ciągi od zera, ale wtedy implementacja jest analogiczna (wystarczy odpowiednio pozmieniać indeksy we wzorze).

Jak dobrać parametry c i p? Analizę modulo p zostawimy na potem, bo dotyczy ona większości hashy. Natomiast c powinno być z p względnie pierwsze i być większe od wielkości alfabetu σ (zwykle używam dwu- lub trzykrotnie większego). Gdyby c miało wspólny dzielnik z p, to odpowiednio dobrane słowo mogłoby doprowadzić do trwałego „wyzerowania" hasha. Należy też uważać, aby $a_i > 0$, bo inaczej słowa złożone z zer często stanowią kontrprzykłady.

2.3 Modyfikacje

Dynamiczny # wielomianowy W przypadku, gdy ciąg a może się zmieniać, możemy skonstruować odpowiednie drzewo przedziałowe (bądź drzewo Fenwicka) pozwalające na aktualizowanie wartości h_i. Używając drzewa przedziałowego najwygodniej jest je zmodyfikować tak, aby w poddrzewie był hash obejmowanego podciągu. Drzewem Fenwicka możemy utrzymywać sumy prefiksowe po ciągu $a'_i = c^{n-i}a_i$, i preprocesować odwrotności c^k, aby móc liczyć h_i.

Pierwsza różnica (`mismatch`) Chcąc znaleźć pierwszą taką pozycję k, na której $a_k \neq b_k$, możemy wykorzystywać wyszukiwanie binarne i hashe. Jest to przydatne, szczególnie, gdy a i b są podsłowami wcześniej znanych, większych ciągów. Producedura działa w czasie $O(\log n)$.

Porównywanie leksykograficzne Korzystając z definicji chcemy porównać pierwszą pozycję, na której ciągi się różnią, więc wykorzystujemy poprzedni algorytm. Dzięki temu podejściu możemy efektywnie rozwiązać problem minimalnej rotacji cyklicznej oraz zbudować tablicę sufiksową (odpowiednio w czasie $O(n \log n)$ i $O(n \log^2 n)$).

2.4 [XVII OI] Korale

Treść Dany jest ciąg $a = (a_1, a_2, ..., a_n)$. Chcemy wybrać parametr $k \geqslant 1$ i wykonać następujący proces: dopóki $|a| \geqslant k$, bierzemy podciąg (naszyjnik) p złożony z pierwszych k elementów a, zapisujemy go i usuwamy z a. Dwa naszyjniki p i q są różne, jeżeli $p \neq q$ i $p \neq q^T$ (gdzie q^T oznacza odwrócony ciąg q). Znajdź taki parametr k, że otrzymamy jak najwięcej *różnych* naszyjników.

Rozwiązanie Przypomnijmy, że $H_n = \frac{1}{1} + \frac{1}{2} + ... + \frac{1}{n} = O(\log n)$. Rozważmy zatem rozwiązanie dla danego parametru k. Policzmy hashe wielomianowe dla a oraz a^T. Teraz, symulując proces odcinania naszyjników, hash naszyjnika to (w pewnym sensie równocześnie) $\mathcal{H}(a[ik : (i+1)k - 1])$ i $\mathcal{H}(a[ik : (i+1)k - 1]^T)$. Oba te hashe powinniśmy utożsamiać ze sobą, ponieważ tak zdefiniowaliśmy równość naszyjników. Wystarczy zatem zapisać jedną z tych dwóch wartości

(ale taką, by zawsze wyznaczyć ją tak samo – np. mniejszą). Wszystkie hashe wrzucamy do jakiejś struktury zbioru, a na koniec sprawdzamy, ile znajduje się w niej unikatowych wartości. Możemy to wszystko zaimplementować w $O(\frac{n}{k} \log \frac{n}{k})$ (ale z dosyć małą stałą), zatem sumaryczna złożoność to $O(n \log^2 n)$.

3 Zbiory

W przypadku zbiorów najczęściej potrzebujemy szybko je porównywać oraz dodawać i usuwać z nich elementy. Weźmy zatem pewien (multi)zbiór $s = \{s_1, s_2, ..., s_n\}$ i spróbujmy opracować hash $\mathcal{H}(s)$. Załóżmy, że s_i są liczbami.

3.1 # Nieuporządkowany

Powszechnie lubianą operacją łączną, przemienną i odwracalną (każda z tych własności daje nam szersze pole działania) jest dodawanie. Moglibyśmy zatem zrobić głupi hash będący sumą elementów s. Pójdźmy krok dalej: każdej liczbie x przyporządkujmy (w miarę losowy) identyfikator $g(x)$ również będący liczbą, i niech

$$\mathcal{H}(s) = \sum_{x \in s} g(x) \mod p$$

Dodawanie i usuwanie elementu x realizujemy przez dodanie lub odjęcie $g(x)$. W oczywisty sposób hash jest niezależny od kolejności elementów, do tego działa on również dla multizbiorów. Dzięki temu, że g jest losowe, trudno jest znaleźć kontrprzykład, i możemy założyć, że cały hash też jest w miarę losowy.

3.2 # Przynależności

W przypadku, gdy mamy do czynienia jedynie ze zbiorem s (czyli elementy s_i są z pewnością parami różne) możemy trochę uprościć nasz hash. Tym razem wykorzystajmy jedną z operacji bitowych, o jakże ciekawych własnościach: `xor`.

Ponownie przyporządkujmy każdej liczbie losowy identyfikator $g(x)$, ale pozbądźmy się arytmetyki modulo p. Hash jest po prostu równy

$$\mathcal{H}(s) = \bigoplus_{x \in s} g(x)$$

Dzięki własnościom xora, zarówno usunięcie jak i dodanie elementu x realizujemy przez przexorowanie hasha przez $g(x)$. Wbrew pozorom, często zmierzamy się zarówno z multizbiorami jak i zbiorami. Ten hash może np. modelować do jakich zbiorów należy dany obiekt. Warto jednak zwrócić uwagę, że `xor` nie współgra dobrze z operacjami arytmetycznymi (praktycznie nie da się obsłużyć np. dodawania 1 do wszystkich elementów zbioru).

Zastanówmy się chwilę, jak dokładnie działa ten hash. Bity na każdej pozycji są niezależne. Gdy losujemy identyfikator, dany bit jest jedynką z prawdopodobieństwem około $\frac{1}{2}$. Zatem prawdopodobieństwo, że bit hasha na danej pozycji

jest jedynką również jest równe około $\frac{1}{2}$ (bo wtedy liczba jedynek na tej pozycji identyfikatorów jest nieparzysta). Zatem ten hash ma bardzo dobre własności pod względem losowości, do tego bardzo łatwo jest go liczyć.

3.3 # Odwracalny

Bardzo ciekawym problemem jest jego wariacja, w której chcemy być w stanie opracować taki hash zbioru, żeby móc łatwo odtworzyć elementy zbioru i do tego łatwo móc łatwo go modyfikować hash (dodając i usuwając elementy zbioru). Zacznijmy od nałożenia ograniczenia, że chcemy odtwarzać dokładnie k elementów zbioru.

Metoda jest dosyć wyjątkowa, i trudno podać sposób, jak na nią wpaść. Pomysł jest następujący: skorzystajmy z wielomianów i dodawania.

Każdy element zbioru x (zakładamy, że jest on liczbą) rozbijamy na ciąg

$$(x^0, x^1, x^2, ..., x^k)$$

To jest właśnie nasz hash zbioru $\{x\}$. Dodając więcej elementow do zbioru, dodajemy potęgi na odpowiednich pozycjach.

Pozostaje odtworzyć elementy $\{x_1, x_2, ..., x_k\}$. Oznaczmy hash jako $h = (h_0, h_1, ..., h_k)$. Warto zauważyć, że wartość h_0 mówi nam, ile elementów znajduje się w zbiorze. Wystarczy teraz zapisać układ równań dla każdego $l = 0, 1, ..., k$:

$$\sum_{i=1}^{k} a_i^l = h_l$$

Rozwiązanie układu równań tej postaci może być trudne, więc możliwe, że np. będziemy musieli skorzystać z tego, że elementy zbioru są całkowite i małe. Dla małych przypadków możemy opracować odpowiednie wzory. Spróbujmy to zrobić dla $k = 2$:

$$\begin{cases} x_1 + x_2 = h_1 \\ x_1^2 + x_2^2 = h_2 \end{cases} \implies x_1^2 + (h_1 - x_1)^2 = h_2 \implies 2x_1^2 - 2h_1x_1 + (h_1^2 - h_2) = 0$$

$$x = \frac{h_1 \pm \sqrt{2h_2 - h_1^2}}{2}$$

Do czego to się może przydać? Zauważmy, że gdybyśmy mieli ciąg pustych zbiorów, i chcieli dodawać do wszystkich zbiorów na przedziale jakiś element, to za pomocą tego hasha sprowadzamy problem do dodawania odpowiednich wartości na przedziale.

3.4 [ONTAK 2018] Ojciec Biteusz

Treść Mamy dany ciąg $s = (s_1, s_2, ..., s_n)$ oraz q operacji ($1 \leqslant s_i \leqslant t$). Pierwszym rodzajem operacji (modyfikacja) jest podmiana danego elementu ciągu.

Drugą operacją (zapytaniem) jest stwierdzenie, czy dla danego podciągu $s[l:r]$ istnieją takie liczby a, b, że każda z liczb $a, a+1, ..., b$ występuje dokładnie raz w tym podciągu.

Rozwiązanie Liczymy (np. drzewem przedziałowym lub drzewem Fenwicka) hashe przynależności na każdym przedziale ciągu. Jeżeli liczby a, b istnieją, to a jest minimum podciągu – zatem w drzewie przechowujemy też minima, do tego musimy mieć $b = a + (r - l)$. Pozostaje stwierdzić, czy hash zbioru elementów podciągu (który liczymy drzewem) jest taki jak hash $\{a, a+1, ..., b\}$. Możemy to zrobić licząc hashe $g_c = \mathcal{H}(\{1, 2, ..., c\})$ (dla każdego c), i ponieważ mamy do czynienia z xorem, bardzo łatwo możemy policzy poszukiwany hash: $g_{a-1} \oplus g_b$. Podsumowując, mamy rozwiązanie w $O(n + t + q \log n)$.

4 Punkty

4.1 # Zbioru punktów

A co, gdy naszymi obiektami są punkty, i do tego chcielibyśmy pozwalać na proste przekształcenia zbioru (np. translację)? Chcemy, aby hash był podobny do klasycznego hasha zbioru z dodawaniem (aby dodawanie w translacji miało sens), ale do tego chcemy, żeby hash pojedynczego punktu był prosty do obliczenia, aby łatwo było wyprowadzić ich przekształcenie (zatem nie możemy go po prostu losować). Takie wymagania realizuje ten hash:

$$\mathcal{H}(P) = \sum_{(x,y) \in P} \alpha^x \beta^y \mod p$$

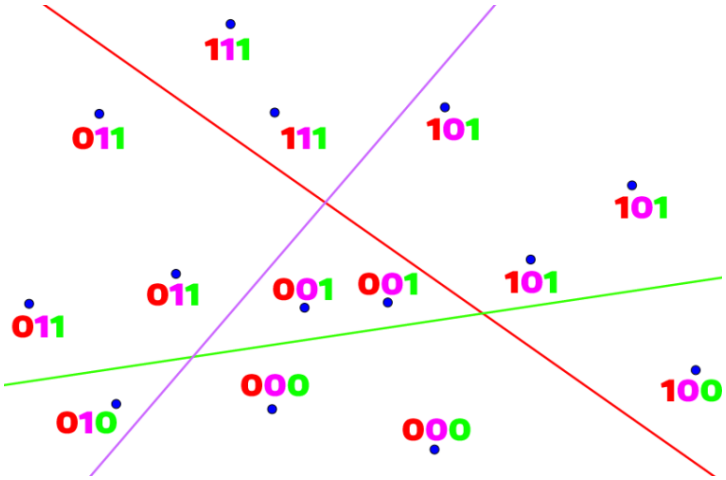
Gdzie $2 \leqslant \alpha, \beta < p - 1$ to podstawy hasha. Podobnie jak hash zbioru, łatwo jest dodawać i odejmować punkty. A jak wykonać translację? Wystarczy wykorzystać własność potęgowania:

$$\mathcal{H}(P + [\Delta x, \Delta y]) = \alpha^{\Delta x} \beta^{\Delta y} \mathcal{H}(P)$$

4.2 # Przez podział prostymi

Nie do końca mamy tutaj do czynienia ze zwykłym haszowaniem, i analiza prawdopodobieństwa nie odnosi się do tej sekcji. W tym wypadku kolizje są nawet pożądane.

Tym razem naszym celem jest pogrupowanie punktów w taki sposób, że punkty leżące blisko siebie są w tej samej grupie. Pomysł jest prosty: podzielmy płaszczyznę w miarę losowy sposób rysując proste, i punkty leżące w tych samych zamkniętych prostymi fragmentach przydzielmy do tej samej grupy. Więcej o tym podejściu można poczytać [tutaj], natomiast jest ono łatwiejsze do zrozumienia po zwizualizowaniu (grafika pochodzi z tego samego bloga):



Przydział do grupy można przeprowadzić przez stwierdzenie dla każdej prostej, z której jej strony leży punkt. Możemy to wygodnie zrobić korzystając z równania prostej $Ax + By + C$ (wartości dodatnie i ujemne leżą na różnych półpłaszczyznach). Proste można losować przez losowanie jej końców spośród zbioru punktów i jej lekkie przesunięcie (dzięki temu proste będą w pobliżu punktów).

Co można z tym zrobić? Punkty leżące blisko siebie prawdopodobnie są w tej samej grupie, więc możemy np. uprościć poszukiwania najbliższego sąsiada do danego punktu. Poza tym, jest to ciekawy pomysł.

5 Kolizje

5.1 Analiza prawdopodobieństwa

Zacznijmy od oznaczenia liczby różnych hashy jako p. Na początek podstawowa własność: porównując jednokrotnie hashe dwóch obiektów, prawdopodobieństwo kolizji to $\frac{1}{p}$.

Sprawa komplikuje się, gdy naraz porównujemy wiele hashy (np. utrzymujących zbiór hashy wielu obiektów) – wtedy wystarcza około $\sqrt{p}$ hashy, aby szansa na kolizję była rzędu 50%. Najczęściej hash 16-bitowy to zdecydowanie zbyt mało, więc zwykle korzystamy z 32-bitowych (bo na 32-bitowym systemie jest to szybsze, i chcemy mieć dostęp do typu 64-bitowego – na wszelki wypadek). W skrajnych sytuacjach możemy spróbować skorzystać z hashy 64-bitowych.

5.2 Hash wielokrotny

A co, gdy szansa kolizji jest zbyt duża, a nie mamy jak zwiększyć liczby możliwych hashy? Wtedy wykorzystujemy hash wielokrotny. Opisujemy nasz hash $\mathcal{H}$ jako para dwóch hashy $\mathcal{H}_1, \mathcal{H}_2$:

$$\mathcal{H}(x) = (\mathcal{H}_1(x), \mathcal{H}_2(x))$$

Wtedy liczba hashy zwiększa się do iloczynu liczby hashy składowych. W ekstremalnych przypadkach możemy powtórzyć ten pomysł n-krotnie, ale w praktyce nigdy się nie przyda się coś więcej niż podwójny hash.

5.3 (Nie)sprytne rozwiązania

5.3.1 Modulo 2^k i słowo Thuego-Morse'a

Wykorzystując hashe wielomianowe moglibyśmy wpaść na pomysł, aby liczyć je modulo 2^{32} lub 2^{64}. Byłby to dobry pomysł, ale niestety istnieje łatwy sposób generowania kolizji, który może się pojawić na konkursach (w szczególności na Olimpiadzie).

Okazuje się, że istnieje specjalna klasa ciągów nazywanych słowami Thuego-Morse'a, dla których pierwsza i druga połowa słowa ma taki sam hash wielomianowy. Tak wyglądają przykładowe iteracje tych słów (są to słowa binarne):

0	0
1	0 1
2	01 10
3	0110 1001
4	01101001 10010110

Ciekawostką jest, że i-ty znak (indeksując od 0) słowa Thuego-Morse'a to

$$T_i = popcount(i) \bmod 2$$

Więcej informacji na ten temat:

`https://codeforces.com/blog/entry/4898`

Wniosek jest taki, że lepiej nie używać podstawy 2^k, chyba że dane są specyficznej postaci lub jako składowa w hashu wielokrotnym (dalej jest to niebezpieczne, ale zwykle nie stanowi problemu).

5.4 Generowanie kolizji

Artykuł o generowaniu kolizji:

`https://codeforces.com/blog/entry/60442`

Chcemy wygenerować dowolną parę o takim samym hashu. Najczęściej możemy wykorzystać wariację *meet in the middle*: generujemy zbiór losowych obiektów wraz z ich hashami, który możemy zmieścić w pamięci, a następnie dowolnie długo generujemy więcej obiektów, sprawdzając, czy dany hash już nie wystąpił. Dzięki paradoksowi dnia urodzin zadziała to dosyć szybko.

6 Bonus: trochę algebry liniowej

A na koniec ciekawy sposób hashowania ciągów, który pozwoli nam utożsamiać ze sobą *rotacje cykliczne*[2] tego samego ciągu. W celu stworzenia tego hasha posłużymy się odrobiną algebry liniowej.

Wprowadźmy pojęcie *śladu* macierzy. Ślad macierzy A oznaczamy $\mathrm{tr}(A)$. Będzie to suma na głównej przekątnej macierzy (przyjmijmy, że są kwadratowe; główna przekątna to ta, która idzie zgodnie z backslashem – \).

My zrobimy następująco: każdemu znakowi przypiszemy losową macierz 2×2 o wartościach modulo p, a hashem ciągu będzie ślad iloczynu wszystkich kolejnych macierzy. Bowiem okazuje się, że taki ślad jest dosyć losowy, jednak prawdziwa jest bardzo przydatna własność: $\mathrm{tr}(AB) = \mathrm{tr}(BA)$ (warto pamiętać, że mnożenie macierzy nie jest przemienne).

Dowód. Rozpiszmy mnożenie macierzy z definicji:

$$X = AB \Longrightarrow X_{i,j} = \sum_k A_{i,k} B_{k,j}$$

$$Y = BA \Longrightarrow Y_{i,j} = \sum_k A_{k,j} B_{i,k}$$

A teraz ślady:

$$\mathrm{tr}\,(X) = \sum_i X_{i,i} = \sum_i \sum_k A_{i,k} B_{k,i}$$

$$\mathrm{tr}\,(Y) = \sum_i Y_{i,i} = \sum_i \sum_k A_{k,i} B_{i,k}$$

Nietrudno zauważyć, że w jednym ze śladów wystarczy zamienić ze sobą nazwy zmiennych i, k (idą po takich samych zakresach), i otrzymamy drugi ze śladów. Czyli teza jest poprawna.

Zauważmy, że fakt w szczególności daje nam $\mathrm{tr}(ABC) = \mathrm{tr}(CAB)$, i w ogólności ślady iloczynu rotacji cyklicznych są sobie równe.

6.1 Realia

No dobra, ale jak faktycznie zrobić taki hash? Piszemy wymaganą obsługę macierzy 2×2 i ich losowanie, spamiętując raz wylosowaną macierz dla danego elementu. Zauważmy, że sam ślad iloczynu macierzy to taki hash „właściwy", a sama macierz będąca iloczynem to hash „częściowy", potrzebny do składania hasha z innymi. Najprawdopodobniej przyda nam się liczenie hasha podsłowa ciągu, więc zastanówmy się, jak to zrobić. Mnożenie macierzy jest operacją łączną, więc możemy wykorzystać drzewo przedziałowe, aby liczyć hash częściowy podsłowa w $O(\log n)$, z konstrukcją w $O(n)$.

[2] Rotacja cykliczna słowa s to dowolne słowo postaci vu, gdzie $s = uv$.

6.2 Ekstrawagancja i więcej algebry

Ten pomysł jest mojego autorstwa i nie mogę podać do niego dobrego źródła. Polecam samodzielnie poszukać informacji na temat kolejnych pojęć, bo sporo tutaj przeskakuję, i spróbować samemu wyprowadzić te rozważania.

Spróbujmy pójść o krok dalej: liczyć hash podsłowa w $O(1)$ z liniową konstrukcją. Taki pomysł podsuwa nam idea sum prefiksowych: co gdyby przechowywać iloczyn macierzy znaków prefiksu ciągu? Możemy tak zrobić, ale teraz, aby policzyć iloczyn dowolnego przedziału, napotkamy pewien problem. Na przykładzie, chcemy z iloczynu $ABCD$ otrzymać CD. Spróbujmy pozbyć się AB macierzą X, którą ją „wycofa" (jest jej odwrotnością).

$$X(ABCD) = CD$$

Okazuje się, że macierz X w przykładzie (czasem) istnieje i jest w istocie *odwrotnością macierzy* AB, czyli $(AB)^{-1}$. Odwrotność nie zawsze istnieje, ale w przypadku pewnej macierzy 2×2 można ją otrzymać w następujący sposób[3]:

$$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{\det A} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Przy czym tajemnicza wartość, przez którą dzielimy, to *wyznacznik macierzy*: dla macierzy 2×2 jest on równy $\det A = ad - bc$. Odwrotność nie istnieje właśnie wtedy, gdy $\det A = 0$. Łatwo się przekonać, że iloczyn macierzy i jej odwrotności daje (z definicji) macierz tożsamościową, czyli swoistą algebroliniową jedynkę (można się przekonać, że $A\mathbb{I} = \mathbb{I}A = A$):

$$\mathbb{I}_{2\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Kończąc wywód, aby uniknąć dzielenia liczb rzeczywistych, chcemy pozostać przy arytmetyce modulo p. Chcemy zatem zapewnić, aby zawsze istniała odwrotność modularna $\det A$, czyli $\gcd(\det A, p) = 1$. Jak zapewnić, aby tak było? Tym razem skorzystamy z własności wyznacznika:

$$\det AB = \det A \det B$$

Ponieważ zawsze nasze macierze to iloczyny macierzy znaków, wystarczy zapewnić, aby wyznaczniki macierzy znaków były względnie pierwsze z p. Jak to zrobić? Cóż, wystarczy losować macierze, dopóki nie będą dobre.

A na sam koniec: Czy może da się do tego wykorzystać modulo przez overflow, czyli 2^{32} lub 2^{64} (po prostu korzystając z typu `unsigned`)? Okazuje się, że tak. Wystarczy, aby wyznaczniki macierzy znaków były nieparzyste (losujemy...), a odwrotności modularne możemy łatwo policzyć szybkim potęgowaniem z pomocą twierdzenia Eulera: odwrotność a modulo 2^k to:

$$a^{-1} \equiv a^{2^{k-1}-1} \pmod{2}^k$$

[3] `https://www.mathsisfun.com/algebra/matrix-inverse.html :)`

6.3 Posłowie: powrót słów Thuego-Morse'a

Pisząc to parę miesięcy po oryginalnej wydanej wersji kartki, należy trochę skomentować opisane tutaj wykorzystanie macierzy. Mianowicie, niestety okazuje się, że **macierze 2 x 2 mają przypadki, w których bardzo łatwo powstają kolizje**. Tymi przypadkami są znane już słowa Thuego-Morse'a. Nie pomaga wykorzystanie modulo innego, niż 2^{32}. Jedynym znanym mi rozwiązaniem jest wykorzystanie macierzy **3 x 3**, które jakkolwiek będą wolniejsze, ale będą działały poprawnie i hash będzie dobrej jakości. Na szczęście możemy dalej wykorzystać modulo 2^{32}.

Wzory na odwrotności nie są w tej sytuacji takie proste, ale dalej istnieją (raczej nie da się ich tak łatwo zapamiętać). Wyznacznik też można policzyć, wzór też jest bardziej skomplikowany, ale jest pewne twierdzenie algebry liniowej, które ułatwia jego zapamiętanie. Sama zasada działania hasha nie zmienia się przy korzystaniu z macierzy 3 x 3. Jednak, co ważne, losowane macierze dalej powinny mieć wyznaczniki nieparzyste, bo inaczej hash będzie niższej jakości.

Zadanie, na którym można łatwo przetestować swoją implementację jest Prefiksufiks z finału XIX OI, dostępny na Szkopule:

`https://szkopul.edu.pl/problemset/problem/`
`oFbHZH1QYy8yYlyN9AezBIZb/site/?key=statement`