Kółko z haszowania

Konspekt, Maciej Matraszek

(Przydatne wiadomości na wstępie: set, sumy prefiksowe, bin search)

0. Modulo – definicja i operacje:
   - 0.1. modulo = reszta z dzielenia n przez m: n = qm + r, r = n – m[n/m] . 9 = 4 * 2 + 1
   - 0.2. Zapis: a mod b (jak w Pascalu), albo jak w C++ a % b.
   - 0.3. Sprawdzanie parzystości: a % 2 == 0
   - 0.4. Łączność: (a + b)% n = ((a%n) + (b%n))%n ← gdy nie jesteśmy pewni kolejności, to warto dać nawiasy
   - 0.5. (a * b)%n = ((a%n) * (b%n)) % n
   - 0.6. Nie dzielimy! (można „cofnąć mnożenie”, ale to wykracza poza zakres tego kółka)
   - 0.7. W wielu zadaniach w których nie ma potrzeby hashowania, trzeba dać odpowiedź modulo.
   - 0.8. Zadanko: parę (a, b) a,b < 1000 przekształcić na jedną liczbę
1. Haszowanie – definicja
   - 1.1. Haszowanie = przekształcenie obiektu (liczby, pary, ciągu znaków) na liczbę z pewnego przedziału.
   - 1.2. Funkcja haszująca – haszuje: H – liczba pierwsza, LL = long long

     h(LL a) = a%H, h(a, b) = (a*10^9 + b)%H.
   - 1.3. Tylko relacja '=', A != B → pewne, A == B → z prawd. 1 – 1/H
   - 1.4. Ale z Paradoksu Dnia Urodzin wiemy, że gdy jest N ~ sqrt H, to prawdopodobieństwo, że istnieją dwa obiekty mające te samo modulo H jest większe od ½
   - 1.5. Jurorzy są świadomi, że rozwiązania będą na haszowaniu. Nie ma się czego bać.
   - 1.6. Fajne liczby pierwsze:
     10^9 + 7, 10^6 + 3, 10^4 + 9, 5 555 553, faktoryzacja w konsoli: factor
2. Problem I: mamy zbiór S, do którego dodawane są elementy, i około |S| zapytań: czy a należy do S?
   - 2.1. Zbiór = set, pojedyncza operacja O(log N). W sumie O(N log N)
     a) Wady: zajmuje dużo pamięci i da się szybciej!
   - 2.2. Haszujemy!
     a) Robimy tablicę booli BSET [H]
     b) Wrzucanie: k = h(a_i), BSET[k] = 1;
     c) Sprawdzanie: k = h(a), if(BSET [k] == 1)
     d) Każda operacja O(1)
     e) Bardzo często jest ok, ale też mamy kolizje gdy N nie jest małe w porównaniu do H...
   - 2.3. Metoda łancuchowa
     a) W tablicy VSET zamiast booli trzymamy vectory obiektów, które mają danego hasza
     b) Wrzucanie: VSET[k] .push_back(a_i)
     c) Sprawdzanie:

        ```
        for(int i = 0, i != size(VSET[k]); ++i) if( VSET[k][i] == a) return true;
        return false;
        ```
     d) Wrzucenie: O(1). Sprawdzanie pesymistycznie O(n), w rzeczywistości zachowuje się O(1)
   - 2.4. Usuwanie do samodzielnego wymyślenia w BSET i VSET.

2.5. Oszacowania i kolizję są PESYMISTYCZNE, jeśli się dobrze dobierze liczbę H, to trzeba mieć strasznego pecha, by ucięło chociaż test.

3. Problem II: Dla liczby a powiedzieć ile razy występuje w S

3.1. Hashowaniem analogicznie.

a) W BSET zamiast 1, robimy ++

b) w VSET trzymamy pary (obiekt, ilość)

3.2. Map = coś jak tablica i set w jednym, dla dowolnego indeksu-klucza coś trzymamy:

a) Nagłówek

#include <map>

b) Tworzymy podobnie jak set:

map< KLUCZ, WARTOŚĆ > MAPA;

c) Zwiększanie krotności:

Dla map< COS, int> MAPA: //np: map<LL, int>

MAPA[a]++. ← gdy jeszcze nie przypisywaliśmy, to mamy zero

d) Odpowiedź: MAPA[a].

e) Złożoność jak set (czasowa i pamięciowa)

4. unordered_set, unordered_map:<- takie coś w STLu napisane właśnie z użyciem łańcuchowej

4.1. Działa haszując obiekty (w set) lub klucze (w map)

a) Samo hashuje: podstawowe typy i stringi.

b) Można napisać funkcję haszującą cokolwiek.

c) Najlepiej napisać sobie zewnętrznego haszera na LL i to podawać.

4.2. Zatem działa sporo szybciej od seta, pojedyncza operacja to oczekiwany czas stały

4.3. Jest deterministyczny (w ramach zewnętrznego hasza)

4.4. Nagłówek i tworzenie:

```
#include <tr1/unordered_set> // lub <tr1/unordered_map>
using namespace tr1;
(Jeśli stary kompilator, to <ext/hash_set> i using …  __gnu_cxx)
unordered_set< TYP> US;
unordered_set <KLUCZ, WARTOŚĆ> UM;

np.: unordered_map<LL, int> UM; UM[ h(a) ]++;
```

4.5. Reszta analogicznie jak set/map.

4.6. Jeśli ktoś się chce dowiedzieć, jak za klucze dobierać dowolne obiekty:

```
Struct MyHasher {
    LL operator()(const TYP& a) const {
        return … //obliczenia
    } }
    unordered_map<KEY, VALUE, MyHasher> UM;
```

5. Przykład:

2 miliony operacji:
N x y → utwórz nowy punkt
T x1 y1 y2 → sprawdź, czy istnieje kwadrat o boku (x1, y1) do (x1, y2), y1 < y2

Bok kwadratu to a = y2 – y1.
Sprawdź na lewo i prawo: (x1 – a, y1) && (x1 – a, y2) lub (x1 + a, y1) && (x1 + a, y2)

6. Zadanie:

   6.1. Wersja pierwsza (Czwórki z KI):
   Mamy zbiór liczb |S| < 5000.
   a_i < 5*10^5.
   Ile jest takich czwórek a,b,c,d w S (mogą się powtarzać), że a + b + c = d?
   np.: 1 2 3 4 → 4: (1,1,1,3), (1,1,2,4), (1,2,1,4) (2,1,1,4)

   6.2. Wersja druga:
   Tak samo, tylko a_i < 10^15

---

7. Rozwiązania:

   7.1. Pierwszy:
   * <=> a + b = d – c
   Generujemy wszystkie pary (a,b). W tablicy trzymamy dla każdej liczby, ile jest takich par (a,b), że a+b = index. Z założenia a+b <= 10^6, więc się zmieści w pamięci.
   Generujemy pary (d, c). Jeśli c > d, to nie ma sensu, bo lewa > 0, w pp. dodajemy do wyniku ile było par a+b równych różnicy (d – c),

   Sortowanie nie wchodzi w grę, bo par (a,b) jest ~ 25 mln...

   7.2. Drugi:
   Zamiast tablicy używamy unordered_map/ lub własnego adresowania łańcuchowego.

Koniec części pierwszej.

Hashowanie tekstów:

1. Funkcja hashująca:

   1.1. char to liczba z przedziału (-127, 127). tzn można ją dodać i mnożyć przez inne. 'a'=97

   s - słowo, n – długość , operacje róbmy na long longach
   $h(s, n) = (s_1 + s_2*x + \ldots + s_{n-1}*x^{n-1}) \% HASH$
   x = liczba pierwsza > | alfabetu |, najlepiej spora jak 10^4+9
   Ale teraz nie będziemy musieli się ograniczac do rozmiaru tablicy, więc pozwólmy wychodzić za zakres. (To tak jakbyśmy modulowali przez zakres LL = 2^64-1)
   $h(s, n) = s_0 + s_1*x + \ldots + s_{n-1}*x^{n-1}$

   Najlepiej liczymy z Hornera: LL HASH=0; for(int i=n-1;i>=0;==i) HASH = HASH*x + s[i]

2. Zakładamy, że jeśli hasze dla tekstu są równe, to one też.

   2.1. Tzn., przy haszowaniu tekstów już liczymy na prawdopodobieństwo...

   2.2. Oczywiście daje to szansę na kolizję, ale jeśli X jest ok, to nie ma problemu

3. Problem I: chcemy zmienić w słowie jeden znak, jak zmienić hash w O(1)?
   Np.: i-ty znak słowa zamienić na znak c ← c to zmienna typu char
   Z właściwości modulo: znak s_i jest pomnożony przez x^i,
   więc można odjąć $s_i x^i$ i dodać ch* x^i, czyli dodać $(ch – s_i) *x^i$.
   Aby to zrobić pamiętamy w tablicy POWX[] wszystkie potęgi x: POWX[i] = x^i
   Tzn: HASH = HASH + (ch - s[i])*POWX[i]

4. Problem II: Chcemy sprawdzić, czy dwa podsłowa tego słowa są równe w O(1).

   4.1. Chcielibyśmy móc wyznaczyć hasz dla podsłowa, tak, że gdyby on był całym słowem, to byłby taki właśnie

   4.2. Przydadzą się tutaj sumy prefiksowe.
   Zapamiętujemy hasze każdego sufiksu (licząc Hornerem)
   LL hasze[n]=0; for(int i=n-1;i>=0;==i) hasze[i] = hasze[i+1]*x + s[i]
   i mamy:
   hasze[0] = s[0] + s[1]*x + s[2]*x^2 + … + s[n-1]*x^{n-1},
   ….
   hasze[n-2] = s[n-2] + s[n-1]*x
   hasze[n-1] = s[n-1] ,
   hasze[n] = 0.

   4.3. Hash podsłowa [a,b] to: hasze[a] – hasze[b+1] * POWX[b-a+1];
   Bo np. dla słowa aaaa:
   hasze[0] = 1 + x + x^2 + x^3
   a hasze[2] = 1 + x, czyli jak pomnożymy hasze[2] * x^2 to, mamy:
   hashe[2] *x^2 = x^2 + x^3

   Zatem hasze[0] – hasze[2]*x^2 = hasze[2] czyli taki jaki był 'aa'.

   np. weźmy słowo 'babab', niech a=1, b=2, i dla łatwości liczenia x = 3
   h('ba') = 2 + 1*x = 5;

   h('bab') = hasze[2] = 2 + 1*x + 2*x^2 = 2 + 3 + 18 = 23
   h('babab') = hasze[0] = 2 + hasze[1] * x = 2 + x*(a + hasze[2]*x) = 2 + 1*x + hasze[2]*x^2
   = 5 + 23*9 = 212

   h('ba') = h('babab') - h('bab')*x^2 = hasze[0] – hasze[1+1]*POWX[1-0+1] =hasze[0] – hasze[2]*x^2 = 5, czyli zgadza się z naszym pierwszym wyliczeniem

5. Problem III: Jak sprawdzić, czy podslowo słowo jest palindromem?

   5.1. Liczymy hashe od tyłu, tzn chcemy, żeby w drugą stronę wyszło to samo, jak byśmy odwrócili słowo:.
   $g(s, n) = s_0 * x^{n-1} + s_1 * x^{n-2} + \ldots + s_{n-1}$.

   5.2. Hasz podsłowa: hasze[b] - hasze[a-1] * POWX[b-a+1]