GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: HASHING

1. Introducción

El problema que queremos resolver es el siguiente, queremos comparar cadenas de manera eficiente. La forma de fuerza bruta de hacerlo es simplemente comparar las letras de ambas cadenas, que tiene una complejidad de tiempo de $O(\text{mín}(n_1, n_2))$ si n_1 y n_2 son los tamaños de las dos cadenas. Queremos hacerlo mejor. La idea detrás de las cadenas es la siguiente: convertimos cada cadena en un entero y las comparamos en lugar de las cadenas. La comparación de dos enteros es una operación $O(1)$.

2. Conocimientos previos

2.1. Sistemas numeración

Un sistema de numeración es un conjunto de símbolos y reglas que permiten representar datos numéricos.

Un sistema de numeración puede representarse como:

$N = S, R$

Donde:

1. N es el sistema de numeración considerado (p.ej. decimal, binario, etc.).
2. S es el conjunto de símbolos permitidos en el sistema. En el caso del sistema decimal son 0,1,...9; en el binario son 0,1; en el octal son 0,1,...7; en el hexadecimal son 0,1,...9, A, B, C, D, E, F.
3. R son las reglas que nos indican qué números y qué operaciones son válidos en el sistema, y cuáles no. En un sistema de numeración posicional las reglas son bastante simples, mientras que la numeración romana requiere reglas algo más elaboradas.

Los sistemas de numeración pueden clasificarse en dos grandes grupos: *posicionales* y *no-posicionales*:

En los **sistemas no-posicionales** los dígitos tienen el valor del símbolo utilizado, que no depende de la posición (columna) que ocupan en el número. Estos son los más antiguos, se usaban por ejemplo los dedos de la mano para representar la cantidad cinco y después se hablaba de cuántas manos se tenía. También se sabe que se usaba cuerdas con nudos para representar cantidad. Tiene mucho que ver con la cardinalidad entre conjuntos. Entre ellos están los sistemas del antiguo Egipto, el sistema de numeración romana, y los usados en Mesoamérica por mayas, aztecas y otros pueblos. Al igual que otras civilizaciones mesoamericanas, los mayas utilizaban un sistema de numeración de raíz mixta de base 20 (vigesimal). También los mayas preclásicos desarrollaron independientemente el concepto de cero (existen inscripciones datadas hacia el año 36 a. C. que así lo atestiguan.).

En los **sistemas de numeración ponderados o posicionales** el valor de un dígito depende tanto del símbolo utilizado, como de la posición que ése símbolo ocupa en el número.

El número de símbolos permitidos en un sistema de numeración posicional se conoce como base del sistema de numeración. Si un sistema de numeración posicional tiene base b significa que disponemos de b símbolos diferentes para escribir los números, y que b unidades forman una unidad de orden superior.

Los sistemas de numeración actuales son sistemas posicionales, que se caracterizan porque un símbolo tiene distinto valor según la posición que ocupa en la cifra.

2.2. Función hash

A las función *hash*, también se les llama funciones picadillo, funciones resumen o funciones de digest. Una función *hash* es un método para generar claves o llaves que representen de manera casi unívoca a un documento o conjunto de datos. Es una operación matemática que se realiza sobre este conjunto de datos de cualquier longitud, y su salida es una huella digital, de tamaño fijo e independiente de la dimensión del documento original. El contenido es ilegible.

Una función hash H es una función computable mediante un algoritmo, actúa como una proyección del conjunto U sobre el conjunto M.

Observa que M puede ser un conjunto definido de enteros. En este caso podemos considerar que la longitud es fija si el conjunto es un rango de números enteros ya que podemos considerar que la longitud fija es la del número con mayor cantidad de cifras. Todos los números se pueden convertir al número especificado de cifras simplemente anteponiendo ceros.

Normalmente el conjunto U tiene un número elevado de elementos y M es un conjunto de cadenas con un número relativamente pequeño de símbolos. Por esto se dice que estas funciones resumen datos del conjunto dominio.

3. Desarrollo

Para la conversión necesitamos una llamada función *hash*. El objetivo de esto es convertir una cadena en un entero, el llamado *hash* de la cadena. La siguiente condición debe mantenerse: si dos cadenas s y t son iguales ($s = t$), entonces sus hashes deben ser iguales (**hash**$(s) =$ **hash**(t)). De lo contrario no podremos comparar cadenas.

Note, la dirección opuesta no tiene que mantenerse. Si los hashes son iguales (**hash**$(s) =$ **hash**(t)), entonces las cadenas no tienen por qué ser iguales. Por ejemplo una función hash válida sería simplemente **hash**$(s) = 0$ para cada s. Ahora, esto es solo un ejemplo estúpido, porque esta función será completamente inútil, pero es una función hash válida. La razón por la que no es necesario mantener la dirección opuesta, porque porque hay muchas cadenas exponenciales. Si solo queremos que esta función hash distinga entre todas las cadenas que consisten en caracteres en minúscula de longitud menor a 15, entonces el hash ya no cabría en un entero de 64 bits (por ejemplo, sin signo largo largo), porque hay muchos de ellos. Y, por supuesto, no queremos comparar enteros largos arbitrarios, porque esto también tendrá la complejidad $O(n)$.

Por lo general, queremos que la función hash asigne cadenas a números de un rango fijo $[0, m)$, luego, comparar cadenas es solo una comparación de dos enteros con una longitud fija. Y, por

supuesto, queremos que hash **hash**$(s) \neq$ **hash**(t) sea muy probable, si $s \neq t$.

Esa es la parte importante que tienes que tener en cuenta. El uso de hash no será 100 % determinísticamente correcto, porque dos cadenas diferentes pueden tener el mismo hash (los hashes chocan). Sin embargo, en una amplia mayoría de problemas donde son utilizados, esto puede ser ignorado de manera segura ya que la probabilidad de que dos hashes diferentes colisionen es muy pequeña. Mas adelante discutiremos algunas técnicas sobre cómo mantener muy baja la probabilidad de colisiones.

3.1. Cálculo del hash de una cadena

La forma buena y ampliamente utilizada de definir el hash de una cadena s de longitud n es:

$$\textbf{hash}(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + ... + s[n-1] \cdot p^{n-1} mod \quad m = \sum_{i=0}^{n-1} s[i] \cdot p^i mod \quad m$$

donde p y m son algunos números positivos elegidos. Se llama una función de hash rodante polinomial.

Es razonable hacer que p sea un número primo aproximadamente igual al número de caracteres en el alfabeto de entrada. Por ejemplo, si la entrada está compuesta solo de letras minúsculas del alfabeto inglés, p = 31 es una buena opción. Si la entrada puede contener letras mayúsculas y minúsculas, entonces p = 53 es una opción posible. El código en este artículo usará p = 31.

Obviamente, m debería ser un número grande, ya que la probabilidad de que dos cadenas aleatorias colisionen es de aproximadamente $\approx \frac{1}{m}$. A veces se elige $m = 2^{64}$, ya que los desbordamientos de enteros de enteros de 64 bits funcionan exactamente igual que la operación de módulo. Sin embargo, existe un método que genera cadenas en colisión (que funcionan independientemente de la elección de p). Así que en la práctica no se recomienda $m = 2^{64}$. Una buena opción para m es un número primo grande. El código en este artículo solo usará $m = 10^9 + 9$. Este es un número grande, pero aún lo suficientemente pequeño para que podamos realizar la multiplicación de dos valores utilizando enteros de 64 bits.

3.2. Colisión

Una colisión ocurre cuando dos valores de entrada diferente generan el mismo resumen. Una función hash debe ser resistente a la colisión.

El mismo *hash* siempre será el resultado de los mismos datos (funciones deterministas), pero la modificación de la información, aunque sea un solo bit dará como resultado un *hash* totalmente distinto.

La idea básica de un valor *hash* es que sirva como una representación compacta de la cadena de entrada. Cuando dos claves se apuntan a la misma dirección o bucket se dice que hay una colisión y a las claves se les denomina sinónimos.

Formas de disminuir el número de colisiones y garantizar el funcionamiento de la función:

- Esparcir los registros.
- Usar memoria adicional.
- Colocar más de un registro en una dirección (Compartimientos).

Muy a menudo, el hash polinomial mencionado anteriormente es lo suficientemente bueno, y no habrá colisiones durante las pruebas. Recuerde, la probabilidad de que ocurra una colisión es solo $\approx \frac{1}{m}$. Para $m = 10^9 + 9$ la probabilidad es $\approx 10^{-9}$, que es bastante baja. Pero note, que solo hicimos una comparación. ¿Qué pasa si comparamos una cadena s con 10^6 cadenas diferentes. La probabilidad de que ocurra al menos una colisión ahora es $\approx 10^{-3}$. Y si queremos comparar 10^6 cadenas diferentes entre sí (por ejemplo, contando cuántas cadenas únicas existen), entonces la probabilidad de que ocurra al menos una colisión ya es ≈ 1. Está bastante garantizado que la función terminará con una colisión y devolverá el resultado incorrecto.

Hay un truco muy fácil para obtener mejores probabilidades. Podemos simplemente calcular dos hashes diferentes para cada cadena (usando dos p diferentes y/o m diferentes, y comparar estos pares en su lugar. Si m es aproximadamente 10^9 para cada una de las dos funciones hash, entonces esto es más o menos equivalente a teniendo una función hash con $m \approx 10^{18}$. Al comparar 10^6 cadenas entre sí, la probabilidad de que ocurra al menos una colisión ahora se reduce a $\approx 10^{-6}$.

4. Implementación

4.1. C++

```
/* Calcula el hash para una cadena cuyo alfabeto es de [a-z]
* Si el alfabeto fuera de [A-Z] cambiar la 'a' por 'A'
* Si el alfabeto fuera de [A-Z,a-z], entonces p=53 y A->1 ... Z->26
* a->27 y z->52 */

long long computeHash(string s) {
   const long long p = 31; const long long m = 1e9 + 9;
   long long hashValue = 0; long long pPow = 1;
   int ncharacter=s.size(); char c;
   for (int i=0;i<ncharacter;i++) {
      c=s[i];
      hashValue = (hashValue + (c - 'a' + 1) * pPow) % m;
      pPow = (pPow * p) % m;
   }
   return hashValue;
}
```

4.2. Java

```
import java.math.BigInteger;
```

```java
import java.util.*;

public class Hashing {

   static final int multiplier = 43;//131
   static final Random rnd = new Random();
   static final int mod1 = BigInteger.valueOf((int) (1e9 + rnd.nextInt((int) 1
       e9))).nextProbablePrime().intValue();
   static final int mod2 = BigInteger.valueOf((int) (1e9 + rnd.nextInt((int) 1
       e9))).nextProbablePrime().intValue();
   long[] p1, p2;
   long[] hash1, hash2;
   int n;

   public Hashing(String s) {
      n = s.length();
      p1 = new long[n + 1];
      p2 = new long[n + 1];
      hash1 = new long[n + 1];
      hash2 = new long[n + 1];

      p1[0] = 1;
      p2[0] = 1;
      for(int i = 0; i < n; i++) {
         hash1[i + 1] = (hash1[i] + s.charAt(i) * p1[i]) % mod1;
         p1[i + 1] = p1[i] * multiplier % mod1;
         hash2[i + 1] = (hash2[i] + s.charAt(i) * p2[i]) % mod2;
         p2[i + 1] = p2[i] * multiplier % mod2;
      }
   }

   public long getHash(int i, int len) {
      return (((hash1[i + len] - hash1[i] + mod1) * p1[n - i - len] % mod1) <<
          32)
       + (hash2[i + len] - hash2[i] + mod2) * p2[n - i - len] % mod2;
   }

   public boolean equals(int i, int j, int len) {
      return getHash(i, len) == getHash(j, len);
   }

   // Ejemplo de uso
   public static void main(String[] args) {
      String s = "abcabc";
      Hashing h = new Hashing(s);
      System.out.println(true == h.equals(0, 3, 3));
      System.out.println(false == h.equals(1, 3, 2));
   }
}
```

5. Complejidad

La implementación del método hash tiene una complejidad $O(n)$ donde n es la longitud de la cadena.

6. Aplicaciones

La idea básica de un valor hash es que sirva como una representación compacta de la cadena de entrada. Por esta razón decimos que estas funciones resumen datos del conjunto dominio. Los algoritmos de hash son útiles para resolver muchos problemas de cadenas.

Alguna de las aplicaciones típicas de Hashing:

1. Algoritmo de Rabin-Karp para la coincidencia de patrones en una cadena en $O(n)$.
2. Cálculo del número de diferentes subcadenas de una cadena en $O(n^2 \log n)$.
3. Cálculo del número de subcadenas palindrómicas en una cadena.
4. Cálculo del número de cadenas duplicadas en un arreglo de cadenas

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando esta técnica:

- Codeforces - C. Registration system
- Codeforces - B. Equivalent Strings
- Codeforces - B. Password
- MOG - D - Password