GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: FUNCIÓN DE PREFIJO. ALGORITMO DE KNUTH-MORRIS-PRATT (*KMP*)

---

1. Introducción

Dada una cadena s de longitud n se define como **función prefijo** un arreglo ϕ de longitud n, donde $\phi[i]$ almacena la longitud del prefijo propio más largo de la subcadena $s[0 \dots i]$ que también es un sufijo de esta subcadena. Un prefijo adecuado de una cadena es un prefijo que no es igual a la cadena misma. Por definición $\phi[0] = 0$. Matemáticamente la definición de la función de prefijo se puede escribir de la siguiente manera:

$$\pi[i] = \max_{k=0 \dots i} \{k : s[0 \dots k-1] = s[i-(k-1) \dots i]\}$$

Por ejemplo, la función de prefijo de la cadena *abcabcd* es $[0,\ 0,\ 0,\ 1,\ 2,\ 3,\ 0]$ y función de prefijo de la cadena *aabaaab* es $[0,\ 1,\ 0,\ 1,\ 2,\ 2,\ 3]$. De como hallar la función prefijo y su aplicaciones en la programación competitiva se abordará en la presente guía.

2. Conocimientos previos

2.1. Autómata (una máquina de estados finitos)

Un autómata finito (AF) o máquina de estado finito es un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida.

Este modelo está conformado por un alfabeto, un conjunto de estados finito, una función de transición, un estado inicial y un conjunto de estados finales. Su funcionamiento se basa en una función de transición, que recibe a partir de un estado inicial una cadena de caracteres pertenecientes al alfabeto (la entrada), y que va leyendo dicha cadena a medida que el autómata se desplaza de un estado a otro, para finalmente detenerse en un estado final o de aceptación, que representa la salida.

2.2. Donald Knuth

Es un reconocido experto en ciencias de la computación estadounidense y matemático, famoso por su fructífera investigación dentro del análisis de algoritmos y compiladores.

2.3. Vaughan Ronald Pratt

Profesor Emeritus en la Universidad Stanford, es un pionero en el campo de informática. Publicando desde 1969, Pratt ha hecho varias contribuciones a áreas fundacionales como algoritmos de búsqueda, algoritmos de ordenación, y tests de primalidad. Más recientemente su búsqueda se ha centrado en el modelado formal de sistemas concurrentes y espacios de Chu. Un patrón de aplicar modelos de áreas diversas de las matemáticas como geometría, álgebra lineal, álgebra abstracta, y especialmente lógica matemática a informática se extiende por su trabajo.

3. Desarrollo

3.1. Algoritmo trivial

Partiendo del problema inicial podemos diseñar un algoritmo que sigue exactamente la definición de función de prefijo el iteraría por cada posible prefijo de la cadena y comparándola con cada posible subcadena de la cadena.

3.2. Algoritmo eficiente

Este algoritmo fue propuesto por Knuth y Pratt e independientemente de ellos por Morris en 1977. Se utilizó como función principal de un algoritmo de búsqueda de subcadenas.

3.2.1. Primera optimización

La primera observación importante es que los valores de la función de prefijo solo pueden aumentar como máximo en uno.

De hecho, de lo contrario, si $\phi[i+1] > \phi[i]+1$, entonces podemos tomar este sufijo que termina en posición $i+1$ con la longitud $\phi[i+1]$ y elimine el último carácter. Terminamos con un sufijo que termina en posición i con la longitud $\phi[i+1]-1$, que es mejor que $\phi[i]$, es decir, obtenemos una contradicción.

La siguiente ilustración muestra esta contradicción. El sufijo propio más largo en la posición i eso también es un prefijo es de longitud 2 , y en la posición $i+1$ es de largo 4 . Por lo tanto la cadena $s_{i-2}\ s_{i-1}\ s_i\ s_{i+1}$ es igual a la cadena $s_{i-2}s_{i-1}s_is_{i+1}$, lo que significa que también las cadenas $s_0\ s_1\ s_2$ son iguales, por lo tanto $\phi[i]$ tiene que ser 3.

$$\overbrace{\underbrace{s_0\ s_1}_{\pi[i]=2}\ s_2\ s_3}^{\pi[i+1]=4}\ \cdots\ \overbrace{s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{\pi[i]=2}\ s_{i+1}}^{\pi[i+1]=4}$$

Por lo tanto, al pasar a la siguiente posición, el valor de la función de prefijo puede aumentar en uno, permanecer igual o disminuir en cierta cantidad. Este hecho ya nos permite reducir la complejidad del algoritmo a O(n^3) , porque en un paso la función de prefijo puede crecer como máximo en uno. En total, la función puede crecer como máximo n pasos, y por lo tanto también sólo puede disminuir un total de n pasos. Esto significa que sólo tenemos que realizar O(n) comparaciones de cadenas y alcanzar la complejidad O(n^2).

3.2.2. Segunda optimización

Vayamos más allá, queremos deshacernos de las comparaciones de cadenas. Para lograr esto, tenemos que utilizar toda la información calculada en los pasos anteriores.

Entonces, calculemos el valor de la función de prefijo ϕ para $i+1$. Si $s[i+1] = s[\phi[i]]$, entonces podemos decir con certeza que $\phi[i+1] = \phi[i]+1$, ya que sabemos que el sufijo en la posición i de

longitud $\phi[i]$ es igual al prefijo de longitud $\phi[i]$. Esto se ilustra nuevamente con un ejemplo.

$$\overbrace{\underbrace{s_0\ s_1\ s_2}^{\pi[i]}\ \overbrace{s_3}^{s_3=s_{i+1}}}^{\pi[i+1]=\pi[i]+1}\ \ldots\ \overbrace{\underbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]}\ \overbrace{s_{i+1}}^{s_3=s_{i+1}}}^{\pi[i+1]=\pi[i]+1}$$

Si este no es el caso, $s[i+1] \neq s[\pi[i]]$, entonces debemos probar con una cadena más corta. Para acelerar las cosas, nos gustaría pasar inmediatamente a la longitud más larga $j < \pi[i]$, de modo que la propiedad del prefijo en la posición i sostiene, es decir $s[0 \ldots j-1] = s[i-j+1 \ldots i]$:

$$\underbrace{\overbrace{s_0\ s_1}^{\pi[i]}}_{j}\ s_2\ s_3\ \ldots\ \underbrace{\overbrace{s_{i-3}\ s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]}}_{j}\ s_{i+1}$$

De hecho, si encontramos tal longitud j , entonces nuevamente solo necesitamos comparar los personajes $s[i+1]$ y $s[j]$. Si son iguales entonces podemos asignar $\pi[i+1] = j+1$. De lo contrario necesitaremos encontrar el valor más grande menor que j , para el cual se cumple la propiedad de prefijo, y así sucesivamente. Puede suceder que esto dure hasta $j = 0$. Si entonces $s[i+1] = s[0]$, asignamos $\phi[i+1] = 1$, y $\phi[i+1] = 0$ de lo contrario.

Entonces ya tenemos un esquema general del algoritmo. La única pregunta que queda es ¿cómo encontramos efectivamente las longitudes para j . Recapitulemos: para la duración actual j en la posición i para el cual se cumple la propiedad del prefijo, es decir $s[0 \ldots j-1] = s[i-j+1 \ldots i]$, queremos encontrar el mayor $k < j$, para el cual se cumple la propiedad del prefijo.

$$\underbrace{\overbrace{s_0\ s_1}^{j}}_{k}\ s_2\ s_3\ \ldots\ \underbrace{\overbrace{s_{i-3}\ s_{i-2}\ s_{i-1}\ s_i}^{j}}_{k}\ s_{i+1}$$

La ilustración muestra que este tiene que ser el valor de $\phi[j-1]$, que ya calculamos anteriormente.

3.2.3. Algoritmo final

Así que finalmente podemos construir un algoritmo que no realice ninguna comparación de cadenas y solo realice O(n) comportamiento. Aquí está el procedimiento final:

- Calculamos los valores del prefijo $\phi[i]$ en un bucle iterando desde $i = 1$ a $i = n-1$ ($\phi[0]$ simplemente se le asigna 0).
- Para calcular el valor actual $\phi[i]$ establecemos la variable j que denota la longitud del mejor sufijo para $i-1$. Inicialmente $j = \phi[i-1]$.
- Prueba si el sufijo de longitud $j+1$ también es un prefijo comparando $s[j]$ y $s[i]$. Si son iguales entonces asignamos $\phi[i] = j+1$, de lo contrario reducimos j a $\phi[j-1]$ y repita este paso.

- Si hemos llegado a la longitud $j = 0$ y todavía no tenemos una coincidencia, entonces asignamos $\phi[i] = 0$ y pasar al siguiente índice $i + 1$.

4. Implementación

La implementación acaba siendo sorprendentemente breve y expresiva. Este es un algoritmo en línea, es decir, procesa los datos a medida que llegan; por ejemplo, puede leer los caracteres de la cadena uno por uno. y procesarlos inmediatamente, encontrando el valor de la función de prefijo para cada carácter siguiente. El algoritmo aún requiere almacenar la cadena en sí y los valores de la función de prefijo calculados previamente, pero si conocemos de antemano el valor máximo M la función de prefijo puede tomar la cadena, solo podemos almacenar $M + 1$ primeros caracteres de la cadena y el mismo número de valores de la función de prefijo.

4.1. C++

4.1.1. Algoritmo trivial

```
vector<int> prefix_function_trivial(string s) {
   int n = (int)s.length();
   vector<int> pi(n);
   for (int i = 0; i < n; i++)
     for (int k = 0; k <= i; k++)
        if (s.substr(0, k) == s.substr(i-k+1, k)) pi[i] = k;
   return pi;
}
```

4.1.2. Algoritmo eficiente

```
vector<int> prefix_function_efficient(string s) {
   int n = (int)s.length();
   vector<int> pi(n);
   for (int i = 1; i < n; i++) {
      int j = pi[i-1];
      while (j > 0 && s[i] != s[j]) j = pi[j-1];
      if (s[i] == s[j]) j++;
      pi[i] = j;
   }
   return pi;
}
```

4.1.3. KMP

```
vector<int> KMP(string sequence, string pattern){
   int n = sequence.size();
```

```
    int m = pattern.size();
    vector<int> matches;
    vector<int> P(m+1);
    for (int i=0;i<m;i++) P[i] = -1;
    for (int i=0, j=-1;i<m;){
        while (j > -1 && pattern[i] != pattern[j]) j = P[j];
        i++; j++; P[i] = j;
    }
    for (int i=0, j=0;i<n;){
        while (j > -1 && sequence[i] != pattern[j]) j = P[j];
        i++; j++;
        if (j == m){
            matches.push_back(i - m);
            j = P[j];
        }
    }
    return matches;
}
```

4.2. Java

4.2.1. Algoritmo trivial

```
public static  int [] prefix_function_trivial(String s) {
    int n = s.length();
    int [] pi = new int[n];
    Arrays.fill(pi,0);
    for (int i = 0; i < n; i++)
        for (int k = 0; k <= i; k++)
            if (s.substring(0, k) == s.substring(i-k+1, k)) pi[i] = k;
    return pi;
}
```

4.2.2. Algoritmo eficiente

```
public static  int [] prefix_function_efficient(String s) {
    int n = s.length();
    int [] pi = new int[n];
    Arrays.fill(pi,0);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s.charAt(i) != s.charAt(j)) j = pi[j-1];
        if (s.charAt(i) == s.charAt(j)) j++;
        pi[i] = j;
    }
    return pi;
}
```

4.2.3. KMP

```
ArrayList<Integer> KMP(String sequence, String pattern){
   int n = sequence.length();
   int m = pattern.length();
   ArrayList<Integer> matches = new ArrayList<Integer>();
   int [] P = new int[m+1];
   for (int i=0;i<m;i++) P[i] = -1;
   for (int i=0, j=-1;i<m;){
      while (j > -1 && pattern.charAt(i) != pattern.charAt(j)) j = P[j];
      i++; j++; P[i] = j;
   }
   for (int i=0, j=0;i<n;){
      while (j > -1 && sequence.charAt(i) != pattern.charAt(j)) j = P[j];
      i++; j++;
      if (j == m){
         matches.add(i - m);
         j = P[j];
      }
   }
   return matches;
}
```

5. Aplicaciones

Dentro de las diferentes aplicaciones que puede tener este algoritmo en los problemas de concursos podemos citar:

- **Buscar una subcadena en una cadena. El algoritmo de Knuth-Morris-Pratt:** La tarea es la aplicación clásica de la función de prefijo. dado un texto t y una cadena s , queremos encontrar y mostrar las posiciones de todas las apariciones de la cadena s en el texto t. Por conveniencia lo denotamos con n la longitud de la cadena s y con m la longitud del texto t. Generamos la cadena $s + \diamond + t$, dónde $\diamond$ es un separador que no aparece ni en s ni en t . Calculemos la función de prefijo para esta cadena. Ahora piense en el significado de los valores de la función de prefijo, excepto las primeras $n + 1$ caracteres (que pertenecen a la cadena s y el separador). Por definición el valor $\pi[i]$ muestra la longitud más larga de una subcadena que termina en posición i que coincide con el prefijo. Pero en nuestro caso esto no es más que el bloque más grande que coincide con s y termina en la posición i. Esta longitud no puede ser mayor que n debido al separador. Pero si la igualdad $\pi[i] = n$ se logra, entonces significa que la cadena s aparece completamente en esta posición, es decir, termina en la posición i. Eso sí, no olvides que las posiciones están indexadas en la cadena. $s + \diamond + t$. Así, si en alguna posición i tenemos $\pi[i] = n$, luego en la posición $i - (n + 1) - n + 1 = i - 2n$ en la cadena t la cadena s aparece. Como ya se mencionó en la descripción del cálculo de la función de prefijo, si sabemos que los valores del prefijo nunca exceden un cierto valor, entonces no necesitamos almacenar la cadena completa y la función completa, sino solo su comienzo. En nuestro caso esto significa que sólo necesitamos almacenar la cadena $s + \diamond$ y

los valores de la función de prefijo para ello. Podemos leer un carácter a la vez de la cadena t y calcular el valor actual de la función de prefijo.

- **Contando el número de apariciones de cada prefijo:** Aquí discutimos dos problemas a la vez. dada una cadena s de longitud n . En la primera variación del problema queremos contar el número de apariciones de cada prefijo $s[0 \dots i]$ en la misma cadena. En la segunda variación del problema otra cadena t se da y queremos contar el número de apariciones de cada prefijo $s[0 \dots i]$ en t .

  Primero resolvemos el primer problema. Considere el valor de la función de prefijo. $\pi[i]$ en una posición i. Por definición significa que el prefijo de longitud $\pi[i]$ de la cadena s ocurre y termina en la posición i, y ya no hay un prefijo que siga a esta definición. Al mismo tiempo, los prefijos más cortos pueden terminar en esta posición. No es difícil ver que tenemos la misma pregunta que ya respondimos cuando calculamos la función de prefijo en sí: dado un prefijo de longitud j ese es un sufijo que termina en la posición i , ¿cuál es el siguiente prefijo más pequeño $< j$ que también es un sufijo que termina en la posición i . Así en la posición i termina el prefijo de longitud $\pi[i]$, el prefijo de longitud $\pi[\pi[i]-1]$, el prefijo $\pi[\pi[\pi[i]-1]-1]$, y así sucesivamente, hasta que el índice sea cero. Por tanto, podemos calcular la respuesta de la siguiente manera:

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++) ans[pi[i]]++;
for (int i = n-1; i > 0; i--) ans[pi[i-1]] += ans[i];
for (int i = 0; i <= n; i++) ans[i]++;
```

  Aquí, para cada valor de la función de prefijo, primero contamos cuántas veces aparece en la matriz π , y luego calcular las respuestas finales: si sabemos que el prefijo de longitud i aparece exactamente $ans[i]$ veces, entonces este número debe sumarse al número de apariciones de su sufijo más largo que también es un prefijo. Al fnal debemos agregar 1 a cada resultado, ya que también necesitamos contar los prefijos originales.

  Consideremos ahora el segundo problema. Aplicamos el truco de Knuth-Morris-Pratt: creamos la cadena $s + \diamond + t$ y calcular su función de prefijo. La única diferencia con la primera tarea es que solo nos interesan los valores de prefijo que se relacionan con la cadena. t , es decir. $\pi[i]$ para $i \geq n + 1$. Con esos valores podemos realizar exactamente los mismos cálculos que en la primera tarea.

- **El número de subcadenas diferentes en una cadena:** Dada una cadena s de longitud n . Queremos calcular el número de subcadenas diferentes que aparecen en él. Resolveremos este problema de forma iterativa. Es decir, aprenderemos, conociendo el número actual de subcadenas diferentes, cómo volver a calcular este recuento añadiendo un carácter al final.

  Digamos que k es el número actual de subcadenas diferentes en s, y agregamos el carácter c hasta el final de s. Obviamente algunas subcadenas nuevas que terminan en c aparecerá. Queremos contar estas nuevas subcadenas que no aparecieron antes.

  Tomamos la cadena $t = s + c$ y invertimos. Ahora la tarea se transforma en calcular cuántos

prefijos hay que no aparecen en ningún otro lugar. Si calculamos el valor máximo de la función de prefijo $\pi_{\max}$ de la cadena invertida t , luego el prefijo más largo que aparece en s es $\pi_{\max}$ largo. Claramente también aparecen en él todos los prefijos de menor longitud.

Por lo tanto, el número de nuevas subcadenas que aparecen cuando agregamos un nuevo carácter c is $|s|+1-\pi_{\max}$. Entonces, para cada carácter agregado podemos calcular el número de subcadenas nuevas en O(n) veces, lo que da una complejidad temporal de O(n^2) en total.

Vale la pena señalar que también podemos calcular el número de subcadenas diferentes agregando los caracteres al principio o eliminando caracteres del principio o del final.

- **Comprimir una cadena:** Dada una cadena s de longitud n. Queremos encontrar el formato *comprimido* más corto. representación de la cadena, es decir, queremos encontrar una cadena t de longitud más pequeña tal que s puede representarse como una concatenación de una o más copias de t.

  Está claro que sólo necesitamos encontrar la longitud de t. Conociendo la longitud, la respuesta al problema será el prefijo de s con esta longitud.

  Calculemos la función de prefijo para s. Usando el último valor del mismo definimos el valor $k = n - \pi[n-1]$. Demostraremos que si k divide n, entonces k será la respuesta, de lo contrario no habrá compresión efectiva y la respuesta es n.

  Dejar n ser divisible por k. Luego la cadena se puede dividir en bloques de la longitud k. Por definición de la función de prefijo, el prefijo de longitud $n-k$ será igual a su sufijo. Pero esto significa que el último bloque es igual al bloque anterior. Y el bloque anterior tiene que ser igual al bloque anterior. Etcétera. Como resultado, resulta que todos los bloques son iguales, por lo tanto podemos comprimir la cadena s a la longitud k.

  Por supuesto, todavía tenemos que demostrar que esto es realmente lo óptimo. De hecho, si hubiera una compresión menor que k , entonces la función de prefijo al final sería mayor que $n-k$. Por lo tanto k es realmente la respuesta.

  Ahora supongamos que n no es divisible por k . Demostramos que esto implica que la longitud de la respuesta es n. Lo demostramos por contradicción. Suponiendo que existe una respuesta y que la compresión tiene longitud p (p divide n). Entonces el último valor de la función de prefijo tiene que ser mayor que $n-p$, es decir, el sufijo cubrirá parcialmente el primer bloque. Consideremos ahora el segundo bloque de la cuerda. Dado que el prefijo es igual al sufijo, y tanto el prefijo como el sufijo cubren este bloque y su desplazamiento entre sí k no divide la longitud del bloque p (de lo contrario k divide n), entonces todos los caracteres del bloque tienen que ser idénticos. Pero entonces la cadena consta de un solo carácter repetido una y otra vez, por lo que podemos comprimirla en una cadena de tamaño 1 , lo que da $k = 1$, y k divide n. Contradicción.

$$\overbrace{s_0\ s_1\ s_2\ s_3}^{p}\ \overbrace{s_4\ s_5\ s_6\ s_7}^{p}$$

$$s_0\ s_1\ s_2\ \overbrace{\underbrace{s_3\ s_4\ s_5\ s_6}_{\pi[7]=5}\ s_7}^{p}$$

$$s_4 = s_3,\ s_5 = s_4,\ s_6 = s_5,\ s_7 = s_6\ \Rightarrow\ s_0 = s_1 = s_2 = s_3$$

- **Construyendo un autómata según la función de prefijo:** Volvamos a la concatenación de las dos cadenas mediante un separador, es decir para las cadenas s y t Calculamos la función de prefijo para la cadena. $s + \# + t$. Obviamente, desde $\#$ es un separador, el valor de la función de prefijo nunca excederá $|s|$. De ello se deduce que basta con almacenar únicamente la cadena $s + \#$ y los valores de la función de prefijo correspondiente, y podemos calcular la función de prefijo para todos los caracteres posteriores sobre la marcha:

$$\underbrace{s_0\ s_1\ \ldots\ s_{n-1}\ \#}_{\text{necesita almacenarlo}}\quad\underbrace{t_0\ t_1\ \ldots\ t_{m-1}}_{\text{no necesita almacenarlo}}$$

  De hecho, en tal situación, conocer al siguiente personaje $c \in t$ y el valor de la función de prefijo de la posición anterior es información suficiente para calcular el siguiente valor de la función de prefijo, sin utilizar ningún carácter anterior de la cadena t y el valor de la función de prefijo en ellos.

  En otras palabras, podemos construir un autómata (una máquina de estados finitos): el estado que contiene es el valor actual del prefijo. función, y la transición de un estado a otro se realizará a través del siguiente carácter.

  Así, incluso sin tener la cadena t , podemos construir tal tabla de transición $(\text{old}_\pi, c) \to \text{new}_\pi$ usando el mismo algoritmo que para calcular la tabla de transición:

```
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            int j = i;
            while (j > 0 && 'a' + c != s[j]) j = pi[j-1];
            if ('a' + c == s[j]) j++;
            aut[i][c] = j;
        }
    }
}
```

  Sin embargo, en esta forma el algoritmo se ejecuta en $O(n^2 26)$ tiempo para las letras minúsculas del alfabeto. Tenga en cuenta que podemos aplicar programación dinámica y utilizar las partes de la tabla ya calculadas. Siempre que nos alejamos del valor j al valor $\pi[j-1]$, en

realidad queremos decir que la transición (j, c) conduce al mismo estado que la transición como $(\pi[j-1], c)$, y esta respuesta ya está calculada con precisión.

```
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            if (i > 0 && 'a' + c != s[i]) aut[i][c] = aut[pi[i-1]][c];
            else aut[i][c] = i + ('a' + c == s[i]);
        }
    }
}
```

Como resultado construimos el autómata en O($26n$) tiempo.

¿Cuándo es útil un autómata así? Para empezar recuerda que usamos la función de prefijo para la cadena $s + \# + t$ y sus valores principalmente para un único propósito: encontrar todas las apariciones de la cadena s en la cuerda t.

Por lo tanto, el beneficio más obvio de este autómata es la aceleración del cálculo de la función de prefijo para la cadena $s+\#+t$. Al construir el autómata para $s+\#$, ya no necesitamos almacenar la cadena s o los valores de la función de prefijo en él. Todas las transiciones ya están calculadas en la tabla.

Pero hay una segunda aplicación, menos obvia. Podemos usar el autómata cuando la cadena t es una cadena gigantesca construida usando algunas reglas. Pueden ser, por ejemplo, cadenas grises o una cadena formada por una combinación recursiva de varias cadenas cortas de la entrada.

Para completar, resolveremos el siguiente problema: dado un número $k \leq 10^5$ y una cadena s de longitud $\leq 10^5$. Tenemos que calcular el número de ocurrencias de s en el k-ésima cadena Gray. Recuerde que las cadenas de Gray se definen de la siguiente manera:

$$\begin{aligned} g_1 &= \text{a} \\ g_2 &= \text{aba} \\ g_3 &= \text{abacaba} \\ g_4 &= \text{abacabadabacaba} \end{aligned}$$

En tales casos, incluso construir la cadena t. Será imposible, debido a su longitud astronómica. El k-ésima cadena Gray es $2^k - 1$ caracteres de largo. Sin embargo, podemos calcular el valor de la función de prefijo al final de la cadena de manera efectiva, conociendo solo el valor de la función de prefijo al principio.

Además del autómata en sí, también calculamos valore $G[i][j]$ el valor del autómata después de procesar la cadena g_i empezando por el estado j. Y además calculamos valores. $K[i][j]$ el

número de ocurrencias de s en g_i, antes durante el procesamiento de g_i empezando por el estado j. De hecho $K[i][j]$ es el número de veces que la función de prefijo tomó el valor $|s|$ mientras realiza las operaciones. La respuesta al problema será entonces $K[k][0]$.

¿Cómo podemos calcular estos valores? Primero los valores básicos son $G[0][j] = j$ y $K[0][j] = 0$. Y todos los valores posteriores se pueden calcular a partir de los valores anteriores y utilizando el autómata. Para calcular el valor de algunos i recordamos que la cuerda g_i consiste en g_{i-1}, el i carácter del alfabeto, y g_{i-1}. Así el autómata pasará al estado:

$$\text{mid} = \text{aut}[G[i-1][j]][i]$$
$$G[i][j] = G[i-1][\text{mid}]$$

los valores para $K[i][j]$. También se puede contar fácilmente.

$$K[i][j] = K[i-1][j] + (\text{mid} == |s|) + K[i-1][\text{mid}]$$

De esta manera podemos resolver el problema de las cadenas Gray y, de manera similar, también una gran cantidad de otros problemas similares. Por ejemplo, exactamente el mismo método también resuelve el siguiente problema: nos dan una cadena s y algunos patrones t_i , cada uno de los cuales se especifica de la siguiente manera: es una cadena de caracteres comunes y puede haber algunas inserciones recursivas de las cadenas anteriores de la forma t_k^{cnt}, lo que significa que en este lugar tenemos que insertar la cadena t_k cnt veces. Un ejemplo de tales patrones:

$$\begin{aligned} t_1 &= \text{abdeca} \\ t_2 &= abc + t_1^{30} + \text{abd} \\ t_3 &= t_2^{50} + t_1^{100} \\ t_4 &= t_2^{10} + t_3^{100} \end{aligned}$$

Las sustituciones recursivas hacen estallar la cadena, de modo que sus longitudes pueden alcanzar el orden de 100^100

Tenemos que encontrar el número de veces que la cadena s aparece en cada una de las cadenas.

El problema se puede resolver de la misma manera construyendo el autómata de la función de prefijo y luego calculamos las transiciones para cada patrón usando los resultados anteriores.

6. Complejidad

En cuanto a las complejidades del los algoritmos implementados en la guía podemos decir que el algoritmo trivial de la función prefijo tiene complejidad temporal de O(n^3) en el caso de

la implementación del algoritmo eficiente su complejidad es O(n) siendo n en ambos casos la cantidad de caracteres de la cadena. En cuanto a la complejidad espacial es O(n) en ambos casos por el uso de un arreglo (π).

En el caso del algoritmo *KMP* su complejidades tanto temporal como espacial es O($n + m$) y O(n) en ese orden siendo n y m las cantidades de caracteres de la cadena y del patrón respectivamente.

7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver aplicando el algoritmo tratado en la guía:

- UVA - 455 Periodic Strings
- UVA - 11022 String Factoring
- UVA - 11452 Dancing the Cheeky-Cheeky
- UVA - 12604 Caesar Cipher
- UVA - 12467 Secret Word
- UVA - 11019 Matrix Matcher
- SPOJ - Pattern Find
- SPOJ - A Needle in the Haystack
- Codeforces - Anthem of Berland
- Codeforces - MUH and Cube Walls
- Codeforces - Prefixes and Suffixes