Algoritmos sobre strings

Víctor Racsó Galván Oyola

21 de mayo de 2021

Índice

Algoritmo de Knuth-Morris-Pratt (KMP)

Concepto

Idea

Aprovechemos los bordes de los prefijos de una cadena lo mejor que podamos.

Concepto

Idea

Aprovechemos los bordes de los prefijos de una cadena lo mejor que podamos.

Ventajas

- Su complejidad es lineal, si queremos buscar una cadena de longitud n en otra de longitud m, toma $O(n+m)$ operaciones hallar las ocurrencias.
- Se puede transformar el proceso para formar un autómata, de manera que la complejidad se vuelve $O(1)$ por posición (la complejidad original es amortizada con la misma cota)

Concepto

Idea

Aprovechemos los bordes de los prefijos de una cadena lo mejor que podamos.

Ventajas

- Su complejidad es lineal, si queremos buscar una cadena de longitud n en otra de longitud m, toma $O(n+m)$ operaciones hallar las ocurrencias.
- Se puede transformar el proceso para formar un autómata, de manera que la complejidad se vuelve $O(1)$ por posición (la complejidad original es amortizada con la misma cota)

Desventajas

A veces el modelamiento del problema puede ser muy complicado si deseamos usar los bordes

Algoritmo - I

Borde

Sea una cadena s, entonces b es un *borde* de s si:

$$b \in \text{Pref}(s) \cap \text{Suff}(s)$$

Donde $\text{Pref}(s)$ es el conjunto de los prefijos de s y $\text{Suff}(s)$ es el conjunto de los sufijos de s.

Algoritmo - I

Borde

Sea una cadena s, entonces b es un *borde* de s si:

$$b \in \text{Pref}(s) \cap \text{Suff}(s)$$

Donde $\text{Pref}(s)$ es el conjunto de los prefijos de s y $\text{Suff}(s)$ es el conjunto de los sufijos de s.

Función de prefijo

Definimos la *función de prefijo* $\pi(i)$ como:

$$\pi(i) = \max_{0 \leqslant j \leqslant i} \{j : s[0, (j-1)] = s[(i-j+1), i]\}$$

En otras palabras, $\pi(i)$ es la longitud del borde propio más largo del prefijo $s[0, i]$.

Algoritmo - I

Borde

Sea una cadena s, entonces b es un *borde* de s si:

$$b \in \text{Pref}(s) \cap \text{Suff}(s)$$

Donde $\text{Pref}(s)$ es el conjunto de los prefijos de s y $\text{Suff}(s)$ es el conjunto de los sufijos de s.

Función de prefijo

Definimos la *función de prefijo* $\pi(i)$ como:

$$\pi(i) = \max_{0 \leqslant j \leqslant i} \{j : s[0, (j-1)] = s[(i-j+1), i]\}$$

En otras palabras, $\pi(i)$ es la longitud del borde propio más largo del prefijo $s[0, i]$.

Pregunta

¿Cómo usamos los bordes?

Algoritmo - II

Encontrar una ocurrencia

Si tenemos dos cadenas s y t, podemos hallar la función de prefijos de la cadena $s + \# + t$, donde $\#$ es un separador que no pertenece a nuestro alfabeto, y cualquier posición que corresponda a una posición de t que tenga $\pi(i) = |s|$ es la última posición de una ocurrencia de s en t.

Algoritmo - II

Encontrar una ocurrencia

Si tenemos dos cadenas s y t, podemos hallar la función de prefijos de la cadena $s + \# + t$, donde $\#$ es un separador que no pertenece a nuestro alfabeto, y cualquier posición que corresponda a una posición de t que tenga $\pi(i) = |s|$ es la última posición de una ocurrencia de s en t.

Relación de los bordes

Trivialmente tenemos que $\pi(0) = 0$, así que consideremos una posición $i > 0$ y que ya hemos calculado $\pi(j)$ para todo $j < i$, entonces un posible borde sería el máximo borde de $i - 1$ agregándole el caracter s_i.
Sea $l = \pi(i - 1)$. Tendremos 2 posibles casos:

- $s[i - l, i]$ es un borde propio de $s[0, i]$, en cuyo caso terminamos con el máximo borde propio posible.
- En caso contrario, debemos probar con el siguiente máximo borde de menor longitud que l, pero ya que $s[i - l, i - 1] = s[0, l - 1]$, podemos usar $\pi(l - 1)$ para definir dicho valor. Nos iremos al paso anterior luego de esto.

Si planteamos la recursión hasta que l se vuelva 0 o nos hayamos detenido, probaremos todos los candidatos y obtendremos correctamente $\pi(i)$.

Algoritmo - III

Código en C++ para hallar la función de prefijo

```
int k = 0;
vector<int> pi(n, 0);
for(int i = 1; i < n; i++){
    while(k > 0 and s[i] != s[k]) k = pi[k - 1];
    if(k < s.size() and s[i] == s[k]) k++;
    pi[i] = k;
}
```

Algoritmo - III

Código en C++ para hallar la función de prefijo

```
int k = 0;
vector<int> pi(n, 0);
for(int i = 1; i < n; i++){
    while(k > 0 and s[i] != s[k]) k = pi[k - 1];
    if(k < s.size() and s[i] == s[k]) k++;
    pi[i] = k;
}
```

Complejidad

La complejidad es lineal: En cada iteración se aumenta k a lo mucho en 1 y en cada iteración del `while` se reduce k en al menos 1. Ya que k no puede ser negativo, la complejidad es igual a $O(n)$ (n aumentos y n reducciones a lo mucho).

Intentemos un problema de la regional 2016

Game of Matchings

Dada una cadena de caracteres s y un patrón numérico a, determinar cuántas subcadenas w de s cumplen con que cada letra se pueda mapear a un número **diferente** de forma que $a_i = mapeo(w[i])$

Intentemos un problema de la regional 2016

Game of Matchings

Dada una cadena de caracteres s y un patrón numérico a, determinar cuántas subcadenas w de s cumplen con que cada letra se pueda mapear a un número **diferente** de forma que $a_i = mapeo(w[i])$

Solución I

Usar todos mapeos posibles y verificar cada substring válido usando KMP.
Complejidad: $O(|\Sigma|! \cdot n)$. Veredicto: TLE

Intentemos un problema de la regional 2016

Game of Matchings

Dada una cadena de caracteres s y un patrón numérico a, determinar cuántas subcadenas w de s cumplen con que cada letra se pueda mapear a un número **diferente** de forma que $a_i = mapeo(w[i])$

Solución I

Usar todos mapeos posibles y verificar cada substring válido usando KMP.
Complejidad: $O(|\Sigma|! \cdot n)$. Veredicto: TLE

Solución II

Analizar todos los substring de longitud $|a|$ y asignar los mapeos a cada primera aparición de letra para finalmente verificar que sea válido o no.
Complejidad: $O(n^2)$. Veredicto: TLE

Intentemos un problema de la regional 2016

Pista I

¿Podemos reestructurar el patrón y la cadena para que dos cadenas con la misma estructura (como aaba y xxyx) tengan la misma secuencia asociada?

Intentemos un problema de la regional 2016

Pista I

¿Podemos reestructurar el patrón y la cadena para que dos cadenas con la misma estructura (como aaba y xxyx) tengan la misma secuencia asociada?

Solución III

Modificaremos cada una de las posiciones con la distancia entre la letra actual y su última aparición (-1 si es que es la primera aparición), de esta manera las secuencias con la misma estructura tendrán la misma secuencia (aaba y xxyx se mapean a $(-1, 1, -1, 2)$).
Solamente tenemos el problema de qué hacer cuando la distancia es mayor que el prefijo que estamos queriendo cubrir (aba tiene como secuencia $(-1, -1, 2)$, pero awaw tiene como secuencia $(-1, -1, 2, 2)$, a pesar de que tenemos awa y waw como patrones similares). Lo que podemos notar del ejemplo entre paréntesis es que para la cadena waw, el valor 2 en la tercera posición en realidad apunta fuera de la cadena, por lo que en realidad debería ser considerado un -1 para esta. Considerando la función $f(x, len)$ que nos da x si $x < len$ y -1 en caso contrario, podremos resolver el problema en $O(n + |s|)$. Veredicto: AC

Hashing

Concepto

Idea

Usemos una función de mapeo para comparar estructuras rápidamente, bajo cierta probabilidad de fallo.

Concepto

Idea

Usemos una función de mapeo para comparar estructuras rápidamente, bajo cierta probabilidad de fallo.

Ventajas

La comparación es eficiente, pues basta con calcular la función hash y luego comparar los resultados, que suelen ser enteros o tuplas de enteros.

Concepto

Idea

Usemos una función de mapeo para comparar estructuras rápidamente, bajo cierta probabilidad de fallo.

Ventajas

La comparación es eficiente, pues basta con calcular la función hash y luego comparar los resultados, que suelen ser enteros o tuplas de enteros.

Desventajas

La aritmética modular hace que la constante que acompaña a la complejidad sea muy pesada, así que no es recomendado usarlo con una complejidad muy ajustada.

Un poco de análisis - I

Función de hash polinomial

Consideremos que podemos mapear las componentes de una estructura a enteros no negativos menores que B, entonces podemos representar de manera única cada estructura posible usando su representación en base B.

$$f(a) = \sum_{i=0}^{n-1} a_i \cdot B^i$$

Si tuviéramos la posibilidad de operar con números infinitamente grandes eficientemente, la mejor forma sería usando una función como esta, la cual es inyectiva para todas las estructuras con $a_{n-1} \neq 0$, es decir, de longitud n. Sin embargo, en la práctica eso no es posible

Un poco de análisis - I

Función de hash polinomial

Consideremos que podemos mapear las componentes de una estructura a enteros no negativos menores que B, entonces podemos representar de manera única cada estructura posible usando su representación en base B.

$$f(a) = \sum_{i=0}^{n-1} a_i \cdot B^i$$

Si tuviéramos la posibilidad de operar con números infinitamente grandes eficientemente, la mejor forma sería usando una función como esta, la cual es inyectiva para todas las estructuras con $a_{n-1} \neq 0$, es decir, de longitud n. Sin embargo, en la práctica eso no es posible

Mejora común

Consideremos la función módulo un entero $M > B$ y B elegido aleatoriamente bajo las condiciones antetiores.

$$f(a) = \sum_{i=0}^{n-1} a_i \cdot B^i \mod M$$

Un poco de análisis - II

Problemática

Ciertamente, la función módulo M ya no es inyectiva, pues por principio del palomar nos basta tomar $M + 1$ elementos diferentes y al menos 2 tendrán la misma función.

Un poco de análisis - II

Problemática

Ciertamente, la función módulo M ya no es inyectiva, pues por principio del palomar nos basta tomar $M + 1$ elementos diferentes y al menos 2 tendrán la misma función.

La pregunta no es si o no, es ¿Con qué probabilidad?

Ahora debemos preguntarnos, si la función no es inyectiva, ¿Cuál es la probabilidad de que dos elementos **diferentes** tengan el mismo resultado luego de aplicarles la función?

Un poco de análisis - II

Problemática

Ciertamente, la función módulo M ya no es inyectiva, pues por principio del palomar nos basta tomar $M+1$ elementos diferentes y al menos 2 tendrán la misma función.

La pregunta no es si o no, es ¿Con qué probabilidad?

Ahora debemos preguntarnos, si la función no es inyectiva, ¿Cuál es la probabilidad de que dos elementos **diferentes** tengan el mismo resultado luego de aplicarles la función?

Analicemos la probabilidad

Sean dos elementos a y b tales que:

$$f(a) = f(b) \mod M \rightarrow \sum_{i=0}^{n-1} a_i B^i = \sum_{i=0}^{n-1} b_i B^i \mod M$$

$$P(x) = \sum_{i=0}^{n-1} (a_i - b_i) B^i = 0 \mod M$$

$P(x)$ es un polinomio de grado menor o igual a $n-1$, así que la probabilidad de que encontremos un b diferente de a pero con misma función hash es $\leqslant \frac{n-1}{M}$.

Intentemos un problema en Codeforces - I

Watto and Mechanism

Dado un conjunto de n cadenas s_i, se nos darán m consultas t_j y se nos pide verificar si existe algún s_k tal que $|s_k| = |t_j|$ y además s_k y t_i difieran en exactamente una posición.

Intentemos un problema en Codeforces - I

Watto and Mechanism

Dado un conjunto de n cadenas s_i, se nos darán m consultas t_j y se nos pide verificar si existe algún s_k tal que $|s_k| = |t_j|$ y además s_k y t_i difieran en exactamente una posición.

Solución I

Usamos una búsqueda lineal hasta encontrar algún s_k que cumpla con la condición.
La complejidad será $O(ml)$, donde l es la suma de longitudes de las n cadenas s_i.
Veredicto: TLE

Intentemos un problema en Codeforces - I

Watto and Mechanism

Dado un conjunto de n cadenas s_i, se nos darán m consultas t_j y se nos pide verificar si existe algún s_k tal que $|s_k| = |t_j|$ y además s_k y t_i difieran en exactamente una posición.

Solución I

Usamos una búsqueda lineal hasta encontrar algún s_k que cumpla con la condición.
La complejidad será $O(ml)$, donde l es la suma de longitudes de las n cadenas s_i.
Veredicto: TLE

Solución II

Usaremos hashing para guardar todas las cadenas que tengan la misma longitud l en un set S_l, luego podemos fijar qué caracter difiere para cada t_j y buscar el hashing de esa nueva cadena en el conjunto correspondiente (el cambio se puede hallar en $O(1)$).
Veredicto: ¿WA?

Solución III

Cuando tengamos un WA usando Hashing (asumiendo que no hay bugs en el código), debemos analizar si la probabilidad de fallo es lo suficientemente pequeña. En este caso, tendremos una probabilidad de fallo de $\frac{2 \cdot 10^6}{10^9+7} \approx 0{,}002$. A pesar de lo pequeña que se ve la probabilidad, para este problema no parece ser suficiente.

Para mejorar la probabilidad y hacerla más pequeña, debemos considerar que la elección del B es independiente, así que podemos tomar dos B_1 y B_2 para hallar funciones hashing f_1 y f_2, volviendo la probabilidad de fallo $(0{,}002)^2 = 0{,}000004$, ahora la comparación de las cadenas no será por un `int`, sino por una tupla `tuple<int, int>`.

Veredicto: AC.

Se puede cambiar el módulo para f_2, pero no es tan necesario.