GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO-Z (*Z-ALGORITHM*)

---

1. Introducción

Supongamos que nos dan una cadena s de longitud n queremos hallar la posición(i) dentro de la cadena y con i distinto de cero tal que que la cadena que se forma a partir de la posición i es la cadena más larga que es al mismo tiempo un prefijo de s y prefijo del sufijo de s a partir de i. A como resolver este problema y las aplicaciones que tiene le dedicaremos la presente guía de aprendizaje.

2. Conocimientos previos

2.1. Prefijo

Un prefijo es una subcadena que comienza al principio de una cadena. Por ejemplo, los prefijos de *ABCD* son *A*, *AB*, *ABC* y *ABCD*.

2.2. Sufijo

Un sufijo es una subcadena que termina al final de una cadena. Por ejemplo los sufijos de *ABCD* son *D*, *CD*, *BCD* y *ABCD*.

3. Desarrollo

En esta guía, para evitar ambigüedades, asumimos el 0 como índice; es decir: el primer carácter de s tiene índice 0 y el ultimo tiene índice $n-1$. Realizada dicha aclaración vamos a proceder a resolver el problema planteado en la introducción de la guía.

Vamos a partir que el algoritmo que obtengamos como resultado le llamaremos *algoritmo Z* o *función Z*. Adicionalmente queremos que el algoritmo sea capaz de generar un arreglo $z$de enteros con capacidad n donde $z[i]$ es la longitud de la cadena más larga que es, al mismo tiempo, un prefijo de s y un prefijo del sufijo de s a partir de i. El primer elemento del algoritmo Z, $z[0]$, generalmente no está bien definido. En esta guía asumiremos que es cero (aunque no cambia nada en la implementación del algoritmo).

Por ejemplo, aquí están los valores del algoritmo Z calculados para diferentes cadenas:

aaaa - $[0,\ 4,\ 3,\ 2,\ 1]$

aaabaab - $[0,\ 2,\ 1,\ 0,\ 2,\ 1,\ 0]$

abacaba - $[0,\ 0,\ 1,\ 0,\ 3,\ 0,\ 1]$

3.1. Algoritmo trivial

Simplemente iteramos a través de cada posición i y actualizar $z[i]$ para cada uno de ellos, a partir de $z[i] = 0$ e incrementándolo siempre que no encontremos una discrepancia (y siempre

que no lleguemos al final de la línea). Por supuesto, esta no es una implementación eficiente. Ahora mostraremos la construcción de una implementación eficiente.

3.2. Algoritmo eficiente

Para obtener un algoritmo eficiente calcularemos los valores de $z[i]$ a su vez de $i = 1$ a $n - 1$ pero al mismo tiempo, al calcular un nuevo valor, intentaremos hacer el mejor uso posible de los valores calculados previamente.

En aras de la brevedad, llamemos **coincidencias de segmentos** a aquellas subcadenas que coinciden con un prefijo de s. Por ejemplo, el valor del algoritmo Z deseada $z[i]$ es la longitud de la coincidencia del segmento que comienza en la posición i (y eso termina en la posición $i + z[i] - 1$).

Para hacer esto, mantendremos **el** $[l, r)$ **los índices del segmento más a la derecha coinciden**. Es decir, entre todos los segmentos detectados nos quedaremos con el que termina más a la derecha. En cierto modo, el índice r puede verse como el *límite* a la que nuestra cadena s ha sido procesado por el algoritmo; todo lo que pase más allá de ese punto aún no se sabe.

Entonces, si el índice actual (para el cual tenemos que calcular el siguiente valor de la función Z) es i , tenemos una de dos opciones:

- $i \geq r$: la posición actual está **fuera** de lo que ya hemos procesado. Luego calcularemos $z[i]$ con el algoritmo trivial (es decir, simplemente comparando valores uno por uno). Tenga en cuenta que al final, si $z[i] > 0$, tendremos que actualizar los índices del segmento más a la derecha, porque está garantizado que el nuevo $r = i + z[i]$ es mejor que el anterior r.
- $i < r$: la posición actual está dentro de la coincidencia del segmento actual $[l, r)$.

  Luego podemos usar los valores z ya calculados para inicializar el valor de $z[i]$ a algo (seguro que es mejor que comenzar desde cero), tal vez incluso a algún número grande.

  Para esto observamos que las subcadenas $s[l \dots r)$ y $s[0 \dots r - l)$ coincidencia. Esto significa que como aproximación inicial para $z[i]$ podemos tomar el valor ya calculado para el segmento correspondiente $s[0 \dots r - l)$, y eso es $z[i - l]$.

  Sin embargo, el valor $z[i - l]$ podría ser demasiado grande: cuando se aplica a la posición i podría superar el índice r. Esto no está permitido porque no sabemos nada sobre los caracteres a la derecha de r pueden diferir de los requeridos.

  Aquí hay un ejemplo de un escenario similar:

  $$s = aaaabaa$$

  Cuando lleguemos a la última posición ($i = 6$), el segmento de coincidencia actual será $[5, 7)$. La posición 6 entonces coincidirá con la posición $6-5 = 1$, para el cual el valor del algoritmo Z es $z[1] = 3$. Obviamente, no podemos inicializar $z[6]$ a 3 , sería completamente incorrecto. El valor máximo al que podríamos inicializarlo es 1 porque es el valor más grande el que

no nos lleva más allá del índice r del segmento de coincidencia $[l, r)$. Por lo tanto, como **aproximación inicial** para $z[i]$ podemos tomar con seguridad:

$$z_0[i] = \text{mín}(r - i,\ z[i - l])$$

Después de haber $z[i]$ inicializado a $z_0[i]$, intentamos incrementar $z[i]$ ejecutando el algoritmo trivial, porque en general, después del extremo r , no podemos saber si el segmento seguirá coincidiendo o no.

Por lo tanto, todo el algoritmo se divide en dos casos, que difieren sólo en el valor inicial de $z[i]$: en el primer caso se supone que es cero, en el segundo caso está determinado por los valores calculados previamente (usando la fórmula anterior). Después de eso, ambas opciones de este algoritmo se pueden reducir a la implementación del el algoritmo trivial, que comienza inmediatamente después de que especificamos el valor inicial. El algoritmo resulta muy sencillo. A pesar de que en cada iteración se ejecuta el algoritmo trivial, hemos logrado avances significativos al tener un algoritmo que se ejecuta en tiempo lineal.

El algoritmo mantiene un rango $[l, r-1]$ tal que $s[l \dots x]$ sea un prefijo de s y sea lo más grande posible. Como sabemos que $s[0 \dots y-x]$ y $s[x \dots y]$ son iguales, podemos usar esta información al calcular los valores z para las posiciones $x+1, x+2, \dots, y$.

En cada posición k, primero verificamos el valor de $z[k-x]$. Si $k + z[k-x] < y$, sabemos que $z[k] = z[k-x]$. Sin embargo, si $k + z[k-x] \geq y$, $s[0 \dots y-k]$ es igual a $s[k \dots y]$, y para determinar el valor de $z[k]$ necesitamos comparar las subcadenas carácter a carácter. Aún así, el algoritmo funciona en tiempo lineal , porque comenzamos a comparar en las posiciones $y-k+1$ y $y+1$.

4. Implementación

4.1. C++

4.1.1. Algoritmo trivial

```
vector<int> z_function_trivial(string s) {
   int n = s.size();
   vector<int> z(n);
   for (int i = 1; i < n; i++) {
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
         z[i]++;
      }
   }
   return z;
}
```

4.1.2. Algoritmo eficiente

```
vector<int> z_function(string s) {
   int n = s.size();
   vector<int> z(n);
   int l = 0, r = 0;
   for(int i = 1; i < n; i++) {
      if(i < r) {
         z[i] = min(r - i, z[i - l]);
      }
      while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
         z[i]++;
      }
      if(i + z[i] > r) {
         l = i;
         r = i + z[i];
      }
   }
   return z;
}

//Otra implementacion
vector<int> z_functionv2(string s) {
   int n = s.size();
   vector<int> z(n);
   int l = 0, r = 0;
   for (int i = 1; i < n; i++) {
      z[i] = max(0,min(z[i-l],r-i+1));
      while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
                        l = i; r = i+z[i]; z[i]++;
                }
        }
        return z;
}
```

4.2. Java

4.2.1. Algoritmo trivial

```
public static int [] z_function_trivial(String s) {
   int n = s.length();
   int [] z = new int [n];
   for (int i = 1; i < n; i++) {
      while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
         z[i]++;
      }
   }
   return z;
}
```

4.2.2. Algoritmo eficiente

```
public static int [] z_function(String s) {
   int n = s.length();
   int [] z = new int[n];
   int l = 0, r = 0;
   for(int i = 1; i < n; i++) {
      if(i < r) {
         z[i] = Math.min(r - i, z[i - l]);
      }
      while(i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
         z[i]++;
      }
      if(i + z[i] > r) {
         l = i;
         r = i + z[i];
      }
   }
   return z;
}

//Otra implementacion
public static int [] z_functionv2(String s) {
   int n = s.length();
   int []  z =new int[n];
   int l = 0, r = 0;
   for (int i = 1; i < n; i++) {
      z[i] = Math.max(0,Math.min(z[i-l],r-i+1));
      while (i+z[i] < n && s.charAt(z[i]) == s.charAt(i+z[i])) {
         l = i; r = i+z[i]; z[i]++;
      }
   }
   return z;
}
```

5. Aplicaciones

Ahora consideraremos algunos usos del algoritmo Z para tareas específicas:

- **Buscar la subcadena:** Para evitar confusiones llamamos t la cadena de texto y p el patrón. El problema es: encontrar todas las apariciones del patrón p dentro del texto t. Para resolver este problema, creamos una nueva cadena $s = p + \diamond + t$, es decir, aplicamos concatenación de cadenas p y t pero también le ponemos un carácter separador $\diamond$ en el medio (elegiremos $\diamond$ de modo que ciertamente no estará presente en ninguna parte de las cadenas p o t). Calcule el algoritmo Z para s. Entonces, para cualquier i en el intervalo $[0; length(t) - 1]$, consideraremos el valor correspondiente $k = z[i + length(p) + 1]$. Si k es igual a $length(p)$ entonces sabemos que hay una ocurrencia de p en el enésima posición de t , de lo contrario no se

produce p en el enésima posición de t . El tiempo de ejecución (y el consumo de memoria) es O($length(t) + length(p)$) .

- **Número de subcadenas distintas en una cadena:** Dada una cadena s de longitud n, cuente el número de subcadenas distintas de s. Resolveremos este problema de forma iterativa. Es decir: conociendo el número actual de subcadenas diferentes, recalcule esta cantidad después de sumarla al final de s un carácter. Entonces deja k ser el número actual de subcadenas distintas de s. Agregamos un nuevo carácter c a s. Obviamente, puede haber algunas subcadenas nuevas que terminen en este nuevo carácter c (es decir, todas aquellas cadenas que terminan con este símbolo y que aún no hemos encontrado) toma una cadena $t = s + c$ e invertirlo (escribir sus caracteres en orden inverso). Nuestra tarea ahora es contar cuántos prefijos de t no se encuentran en ningún otro lugar de t . Calculemos el algoritmo Z de t y encontrar su valor máximo z_{max} Obviamente, t prefijos de longitud z_{max} también ocurre en algún lugar en medio de t. Claramente, también aparecen prefijos más cortos. Entonces, hemos descubierto que el número de subcadenas nuevas que aparecen cuando el símbolo c se adjunta a s es igual a $length(t) - z_{max}$. En consecuencia, el tiempo de ejecución de esta solución es O(n^2) para una cuerda de longitud n. Vale la pena señalar que exactamente de la misma manera podemos recalcular, todavía en O(n) tiempo, el número de subcadenas distintas al agregar un carácter al principio de la cadena, así como al eliminarlo (desde el final o el principio).
- **Compresión de cadenas:** Dada una cadena s de longitud n. Encuentre su longitud *comprimida* más corta. representación, es decir: encontrar una cadena t de longitud más corta tal que s puede representarse como una concatenación de una o más copias de t . Una solución es: calcular el $algoritmo - Z$ de s , recorrer todos i tal que i divide n. Para en la primera i tal que $i + z[i] = n$. Entonces, la cuerda s se puede comprimir a la longitud i.

6. Complejidad

Una vez visto las dos ideas algorítmicas e implementadas es evidente que la idea trivial que fue la primera analizada su complejidad de O(n^2) siendo n la cantidad de caracteres de la cadena. Mientras tanto la complejidad de la segunda idea es aunque no lo parezca O(n). Lo cual vamos a demostrar a continuación.

La demostración es muy sencilla. Vamos a concentrarnos en el bucle anidado ***while*** , ya que todo lo demás es sólo un grupo de operaciones constantes que se resumen en O(n).

Veremos que en cada iteración del bucle ***while*** aumentará el límite derecho r del segmento. Para ello, consideraremos las variantes del algoritmo.

- $i \geq r$: En este caso, el bucle ***while*** no realizará ninguna iteración (si $s[0] \neq s[i]$), o serán necesarias algunas iteraciones, comenzando en la posición i, cada vez moviendo un carácter hacia la derecha. Después de eso, el límite derecho r será necesariamente actualizado. Entonces hemos encontrado que, cuando $i \geq r$, cada iteración del ***while*** bucle aumenta el valor del nuevo índice r.

- $i < r$:En este caso inicializamos $z[i]$ a un cierto valor z_0 dado por la fórmula anterior. Comparemos este valor inicial z_0 al valor $r - i$. Tendremos tres casos:
  - $z_0 < r - i$: Demostramos que en este caso no se realizará ninguna iteración del ***while*** bucle. Es fácil demostrarlo, por ejemplo, por contradicción: si el ***while*** bucle hiciera al menos una iteración, significaría esa aproximación inicial $z[i] = z_0$ fue inexacto. Pero desde $s[l \dots r)$ y $s[0 \dots r - l)$ son iguales, esto implicaría que $z[i - l]$ tiene el valor incorrecto (menos de lo que debería ser). Así, desde $z[i - l]$ es correcto y es menor que $r - i$, se deduce que este valor coincide con el valor requerido $z[i]$.
  - $z_0 = r - i$: En este caso, el bucle ***while*** puede realizar algunas iteraciones, pero cada una de ellas conducirá a un aumento en el valor del r índice porque comenzaremos a comparar desde $s[r]$, que irá más allá del intervalo $[l, r)$.
  - $z_0 > r - i$: Esta opción es imposible, por definición de z_0.

Entonces, hemos demostrado que cada iteración del bucle interno hace que r avance del puntero hacia la derecha. Desde r no puede ser más que $n - 1$, esto significa que el bucle interno no hará más de $n - 1$ iteraciones. Como el resto del algoritmo obviamente funciona en O(n) , hemos demostrado que todo el algoritmo para calcular funciones Z se ejecuta en tiempo lineal.

7. Ejercicios

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

- CSES - String Matching
- CSES - Finding Borders
- UVA - 455 Periodic Strings
- UVA - 11022 String Factoring
- UVa 11475 - Extend to Palindrome
- Codeforces - Password
- Codeforces - Prefixes and Suffixes
- Codechef - Chef and Strings
- eolymp - Blocks of string
- LA 6439 - Pasti Pas!