Função Z

Daniel Saad

27 de abril de 2026

A função Z é uma função que, dada uma string S de comprimento n, retorna o tamanho da maior substring $S[i, n-1]$. O vetor Z que representa a função Z para a string S é definido como:

$$Z[i] = \max\{k \mid S[i, i+k-1] = S[0, k-1]\} \tag{1}$$

Para string $S = xyxxyxxyxxyxxyx$, o vetor Z é mostrado na Tabela 1. Por exemplo, $Z[4] = 6$ porque a substring $S[4, 9]$ é igual a $S[0, 5]$ e não existe nenhuma substring maior, iniciando na posição 4 que seja igual a uma substring iniciando na posição 0.

Tabela 1: Vetor Z para a string $S = xyxxyxxyxxyxxyx$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$S[i]$	x	y	x	x	y	x	x	y	x	x	y	x	x	y	x
$Z[i]$	15	0	1	0	6	0	1	0	6	0	1	0	3	0	1

Uma forma ingênua de computar o vetor Z é, para cada posição i, comparar a substring $S[i \dots n-1]$ com a substring $S[0 \dots n-1]$ e contar quantos caracteres são iguais, como disposto no Algoritmo 1. Esse processo, no pior caso, leva tempo $\Theta(n-i)$ para cada posição i, então, se computarmos para todo i, gastaremos tempo:

$$\begin{aligned}
&\sum_{i=0}^{n-1} \Theta(n-i) = \\
&\Theta\left(\sum_{i=0}^{n-1} n-i\right) = \\
&\Theta\left(\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i\right) = \\
&\Theta\left(n^2 - \frac{n(n-1)}{2}\right) = \\
&\Theta(n^2)
\end{aligned}$$

Contudo é possível computar o vetor Z em tempo $\Theta(n)$ se reaproveitarmos resultados já computados.

Algorithm 1: NAIVE-Z(S)

Input: $S[0, n-1]$
Output: $Z[0, n-1]$

```
1 for( i ← 0 to n − 1 )
2     Z[i] ← 0
3     while S[Z[i]] = S[i + Z[i]] do
4         Z[i] ← Z[i] + 1
5 return Z
```

Computação eficiente do vetor Z

Suponha que queremos computar $Z[i]$ e que já tenhamos o valor de $Z[k]$ computado para $0 \leq k < i$. Também suponha que tenhamos o intervalo $[l, r]$ tal que $l < i \leq r$ e $r - l + 1 = Z[l]$, em que r é o maior índice tal que $S[l, r]$ é igual a $S[0, r - l]$. Com essas informações, podemos computar $Z[i]$ da seguinte forma:

1. Se $Z[i - l] < r - i + 1$, então $Z[i] = Z[i - l]$.
2. Se $Z[i - l] \geq r - i + 1$, então $Z[i] \geq r - i + 1$. Nesse caso, podemos comparar os caracteres que estão fora do intervalo $[l, r]$ para concluir o valor de $Z[i]$.

Pela definição do intervalo $[l, r]$, temos que $S[l, r] = S[0, r - l] = S[0, Z[l] - 1]$.

No primeiro caso, quando $Z[i-l] < r-i+1$, significa que a substring $S[i, i+Z[i-l]-1]$ é igual a $S[0, Z[i-l]-1]$. Além disso, conseguimos concluir que $S[i+Z[i-l]] \neq S[Z[i-l]]$ porque, se fossem iguais, então $Z[i-l]$ seria maior, contradizendo a computação de $Z[i-l]$. Portanto, $Z[i]$ é igual a $Z[i - l]$.

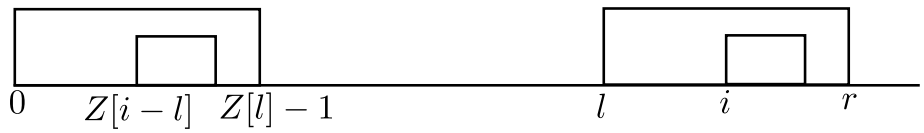

Figura 1: Caso 1: $Z[i - l] < r - i + 1$.

No segundo caso, quando $Z[i - l] \geq r - i + 1$, significa que a substring $S[i, i + r - i] = S[i, r]$ é igual a $S[0, r - l]$. Contudo, nada podemos concluir ainda sobre os caracteres que estão além de r. Nesse caso, podemos comparar os caracteres que estão fora do intervalo $[l, r]$ explicitamente para concluir o valor de $Z[i]$. A Figura 2 ilustra esse caso. Nesse segundo caso, após a computação de $Z[i]$, podemos atualizar o intervalo $[l, r]$ para $[i, i + Z[i] - 1]$.

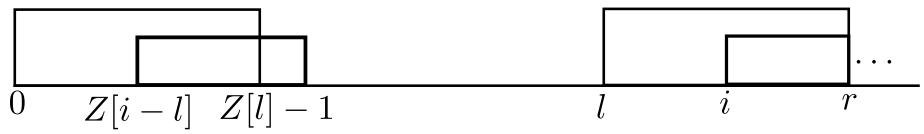

Figura 2: Caso 2: $Z[i - l] \geq r - i + 1$.

Juntando os casos 1 e 2, temos o Algoritmo 2 para computar o vetor Z.

Algorithm 2: Z-ARRAY(S)

Input: $S[0, n-1]$
Output: $Z[0, n-1]$

```
1  Z[0] ← n
2  l ← 0
3  r ← 0
4  for( i ← 1 to n − 1 )
5      if( i ≤ r )
6          Z[i] ← min(r − i + 1, Z[i − l])
7      while i + Z[i] < n e S[Z[i]] = S[i + Z[i]] do
8          Z[i] ← Z[i] + 1
9      if( i + Z[i] − 1 > r )
10         l ← i
11         r ← i + Z[i] − 1
12 return Z
```

Análise

O algoritmo compara cada caractere no máximo uma vez, visto que no primeiro caso, $Z[i]$ é computado sem comparações e no segundo caso, os caracteres comparados são aqueles que estão além do intervalo $[l, r]$, fazendo com que o novo intervalo $[l, r]$ abranja os caracteres comparados com sucesso. Portanto, o tempo gasto para computar o vetor Z é $\Theta(n)$. Em outras palavras, o laço de dentro não pode executar mais de n vezes.

Aplicações

O vetor Z pode ser utilizado em várias aplicações relacionadas a casamento de padrão e análise de periodicidade em strings.

Casamento de Padrões

Suponha que queremos encontrar todas as ocorrências de um padrão $P[0, m-1]$ em um texto $T[0, n-1]$. O truque é criar uma string $P\$T$, em que \$ é um caractere que não aparece em nenhuma das duas strings. As ocorrências de P em T correspondem às posições $i - m - 1$ para as quais $Z[i] = m$. O vetor Z para a string $P\$T$ pode ser computado em tempo $\Theta(m + n)$, o que torna esse método eficiente para encontrar todas as ocorrências de um padrão em um texto.

Tomando o exemplo da string $T = xyxxyxxyxxyxxy$ da Tabela 1 e o padrão $P = yx$ temos o seguinte vetor Z para a string $S = P\$T = yx\$xyxxyxxyxxyxxy$:

Tabela 2: Vetor Z para a string $P\$T = yx\$xyxxyxxyxxyxxy$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$S[i]$	y	x	\$	x	y	x	x	y	x	x	y	x	x	y	x	x	y
$Z[i]$	0	0	0	0	2	0	0	2	0	0	2	0	0	2	0	0	1

Casamento de Padrões Cíclicos

Uma string $P[0, m-1]$ é dita um padrão cíclico de uma string $T[0, n-1]$ se existe algum $0 \leq i < n$ tal que $P[j] = T[(i+j) \mod n]$ para todo $0 \leq j < m$. Por exemplo se $P = xyz$ e $T = zyyxxy$, então P é um padrão cíclico de T a partir da posição $i = 4$, visto que $P[0] = T[4]$, $P[1] = T[5]$ e $P[2] = T[6]$.

Para encontrar todas as ocorrências de um padrão cíclico P em um texto T, podemos criar a string $P\$TT$, onde \$ é um caractere que não aparece em nenhuma das duas strings. O tempo total permanece $\Theta(m+n)$ visto que apenas adicionamos uma cópia de T à entrada, que agora tem tamanho $2n + m + 1$.