i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
s[]	a	a	a	a	a	b	z	a	a	a	a	a	a	b	y
z[]	0	4	3	2	1	0	0	**5**	**6**	4	3	2	1	0	0

Z[i] = tamanho do maior prefixo comum entre **s** e o sufixo que começa em **i, s[i, n - 1];**

Naive O(n^2):

```
vector<int> naive_z_function(const string &s) {
    int n = s.size();
    vector<int> z(n, 0);
    for(int i = 1; i < n; i++) {
        while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
    }
    return z;
}
```

Z function naive : O(n ^ 2)

Ideia naive: para cada sufixo começando em i = 1 até i = n - 1, inicia-se z[i] = 0. Podemos aumentar o tamanho atual de z[i] se e somente se (i + z[i]) < n e o caractere atual sendo comparado de fato bate com o prefixo respectivo: s[i + z[i]] == s[z[i]];

Z-function

A ideia central do algoritmo guarda a essência do algoritmo naive, mas com a definição fundamental dos ponteiros **[L, R]**.

Considere que estamos interessados em calcular **z[i]** para algum **i > 0**.
Definimos **R**: O índice do fim do sufixo de **S**, que termina mais à direita possível de **i**, e que também é prefixo de S. Este sufixo deve começar ESTRITAMENTE ANTES de i.

Definimos **L:** O índice do começo do de **S**, que termina mais à direita possível de **i** e que também é prefixo de S. Este sufixo deve começar ESTRITAMENTE ANTES de i.

Portanto temos: Para um dado **i**, **R = max R** tal que **1 <= L < i <= R <= n - 1**, **S[L, R]** é um prefixo de **S.**
Por que os ponteiros [L, R] são úteis?

A explicação a seguir resolve o problema de calcular **z[i]** para o **i-ésimo** índice da string. É assumido que calculamos corretamente todos os valores de **z[j < i]** e os valores adequados de **L, R**, **1 <= L, R <= n - 1**, anteriores à i.

Caso L < i <= R
O primeiro caso é quando **1 <= L < i <= R**.

A primeira observação é que a substring **S[L, R]**, **L > 0**, é um prefixo de tamanho **R - L + 1,** ou seja a string **S[L, R]** é igual a string **S[0, R - L].**

A segunda observação importante é que como **L < i <= R**, a string **S[i, R]** também já apareceu anteriormente, pois ela está dentro de **S[L, R]**, que já apareceu antes. Portanto, podemos aproveitar essa resposta para calcular **z[i]**.

i	0	1	2	3	4	5	6	7	i = 8	9
s[]	a	b	a	c	a	b	a	b	a	c
z[]	0	0	1	0	3	0	4	0	1	0
							L = 6			**R = 9**

Tabela: exemplo da segunda observação:
Considere que estamos interessados em calcular z[i = 8], i = 8. Temos que L = 6, R = 9, e portanto a substring s[8, 9] = "ac" já apareceu antes em s[2, 3] = "ac", portanto z[8] = z[2] = 1.

Note que NÃO necessariamente a resposta final de **z[i]** vai ser **min(R - L + 1, z[i - L])**, pois ainda podemos aumentar **z[i]** caso **S[i + z[i]] == S[z[i]]**.,
Precisamos da função **min()** porque pode ser que **z[i - L] < (R - L + 1)**.
Por exemplo:

i	0	1	2	3	4	5	6	7	8
s[]	a	a	a	b	x	a	a	a	b
z[]	0	2	1	0	0	4	2 **(Caso z[i - L] < (R - L + 1)**	1	0
						L = 5 R = 8			

Tabela: exemplo de como z[i - L] pode ser MENOR que (R - L + 1)

No caso de **S[i + z[i]] == S[z[i]]** depois de fazer **z[i] = min(R - L + 1, z[i - L])** é necessário continuar calculando **z[i]** conforme o algoritmo naive, e atualizar o valor de **[L, R]** conforme o caso.
Segue um exemplo de onde é necessário continuar o cálculo de **z[i]**.

i	0	1	2	3	4	5	6	7
s[]	a	a	b	a	a	a	b	d
z[]	0	1	0	2	3	1	0	0
				L = 3 R = 4	Caso z[i] > (R - L + 1)			

Tabela: exemplo de como z[i] pode ser MAIOR que (R - L + 1)

Exemplo:

Caso R > i.
Isso quer dizer que não existe nenhum sufixo começando em **1 <= L < i** e terminando em **R >= i**. Neste caso, calculamos **z[i]** da forma naive. Após isso, **L = i, R = i + z[i] - 1**

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

Complexidade

O(n)

Aplicações

Aplicação 1:
String matching: dado **text** e **pat** queremos saber o número de ocorrêcencias de **pat** em **text**.
Fazemos **z = z_function(pat + "#" + text)** onde **'#'** é uma letra que não pertece a pat e nem a text.
As ocorrências de **pat** em **text** se dão onde **z[i] = |pat|**.

Aplicação 2
Matching com caracteres contíguos diferentes**:**
Neste problema ao compararmos duas strings de tamanhos iguais, podemos considerar "match" se houveram até **k** mismatches tais que os **k** mismatches ocorreram de forma contígua.

Ex:
AABC
AXBC (mismatch de 1)

AAZE
XXZE (mismatch de 2)

ABCD
DCBA (mismatch de 4)

Solução

Montemos
Z = z_function(pat + "#" + text);
Z_rev = z_function(rev(pat) + "#" + rev(text));

Para saber o mismatch contíguo entre **pat** e a **text[i]**, **i <= n - m**, basta fazer:
Mismatch = m - (Z[i + m + 1] - z_rev[n - i + 1]

Referências

https://github.com/edsomjr/TEP/blob/master/Strings/text/Algoritmos_de_Busca.md
http://e-maxx.ru/algo/z_function (utilize https://translate.yandex.com/translate para traduzir)
http://codeforces.com/blog/entry/3107