

# Strings.

Material de referencia.

## Índice

<b>1. Arreglo Z.</b>	<b>2</b>
1.1. Implementación . . . . .	2
<b>2. Aho Corasick.</b>	<b>3</b>
2.1. Implementación . . . . .	3
<b>3. Arreglo de sufijos.</b>	<b>5</b>
3.1. Implementación . . . . .	5

## 1. Arreglo Z.

Consideremos un string  $s$ . El arreglo Z de  $s$  es un arreglo de enteros que guarda la longitud del substring más largo que empieza en la posición  $i$  y es prefijo de  $s$ .

En particular, podemos encontrar todas las ocurrencias de un patrón  $P$  en un texto  $T$  calculando el arreglo Z del string  $P + \diamond + T$ , donde  $\diamond$  es algún carácter que no aparece ni en  $P$  ni en  $T$ .

### 1.1. Implementación

Complejidad:  $O(|P| + |T|)$ .

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4
5 #define maxn 100000 //Longitud maxima de los strings.
6
7 string text, pattern, str; //Texto, patron a buscar y string auxiliar.
8 int Z[maxn];             //Arreglo Z.
9
10 //Construye el arreglo Z de str.
11 void buildZ() {
12     int l = 0, r = 0;
13     for (int i = 1; i < str.size(); ++i) {
14         Z[i] = 0;
15         if (i <= r)
16             Z[i] = min(r - i + 1, Z[i - 1]);
17         while (i + Z[i] < str.size() && str[Z[i]] == str[i + Z[i]])
18             ++Z[i];
19         if (i + Z[i] - 1 > r) {
20             l = i;
21             r = i + Z[i] - 1;
22         }
23     }
24 }
25
26 int main() {
27     ios_base::sync_with_stdio(0); cin.tie();
28     //Lee el texto y los patrones.
29     cin >> text >> pattern;
30     str = pattern + '$' + text;
31
32     //Imprime todas las ocurrencias.
33     buildZ();
34     for (int i = 0; i < text.size(); ++i)
35         if (Z[i + pattern.size() + 1] == pattern.size())
36             cout << "Patron encontrado en la posicion " << i << '\n';
37
38     return 0;
39 }
```

Entrada	Salida
AABAACAADAABAABA	Patron encontrado en la posicion 0
AABA	Patron encontrado en la posicion 9
	Patron encontrado en la posicion 12

## 2. Aho Corasick.

Consideremos un texto  $T$  y una lista de patrones  $P_1, \dots, P_n$ . Queremos encontrar todas las ocurrencias de estos patrones en el texto.

El algoritmo de Aho Corasick resuelve este problema implementando un Trie sobre los patrones. Adicionalmente, cada vértice posee una nueva arista que representa el sufijo más largo del camino actual que es prefijo de algún patrón.

### 2.1. Implementación

Complejidad:  $O(|T| + |P_1| + \dots + |P_n| + \#Ocurrencias)$ .

```

1  #include <iostream>
2  using namespace std;
3
4  #define maxc 26      //Longitud del alfabeto.
5  #define maxn 100     //Maximo numero de patrones.
6  #define maxs 100000 //Maximo numero de nodos.
7
8  int n;                //Numero de patrones.
9  string text, pattern[maxn]; //Texto y lista de patrones.
10
11 struct node {
12     node *nxt[maxc]; //Nodo siguiente en el Trie.
13     node *link;      //Sufijo propio mas largo que es prefijo de un patron.
14     bool isEnd[maxn]; //Es nodo terminal de algun patron.
15 };
16
17 int nnodes;          //Numero de nodos.
18 node Trie[maxs];     //Nodos del Trie.
19
20 //Retorna el nodo siguiente.
21 node *nextNode(node *curr, char c) {
22     if (curr == NULL)
23         return Trie;
24     if (curr->nxt[c] == NULL)
25         return nextNode(curr->link, c);
26     return curr->nxt[c];
27 }
28
29 //Construye los links de cada nodo.
30 void buildLink(node *curr) {
31     for (char c = 0; c < maxc; ++c) {
32         node *nxt = curr->nxt[c];
33         if (nxt != NULL) {
34             nxt->link = nextNode(curr->link, c);
35             for (int i = 0; i < n; ++i)
36                 if (nxt->link->isEnd[i])
37                     nxt->isEnd[i] = true;
38             buildLink(nxt);
39         }
40     }
41 }
42
43 //Construye el Trie de patrones.

```

```
44 void buildTrie() {
45     for (int i = 0; i < n; ++i) {
46         node *curr = Trie;
47         for (int j = 0; j < pattern[i].size(); ++j) {
48             char c = pattern[i][j] - 'a';
49             if (curr->nxt[c] == NULL)
50                 curr->nxt[c] = Trie + (++nnodes);
51             curr = curr->nxt[c];
52         }
53         curr->isEnd[i] = true;
54     }
55     buildLink(Trie);
56 }
57
58 int main() {
59     ios_base::sync_with_stdio(0); cin.tie();
60
61     //Lee el texto y los patrones.
62     cin >> text >> n;
63     for (int i = 0; i < n; ++i)
64         cin >> pattern[i];
65     buildTrie();
66
67     //Imprime todas las ocurrencias.
68     node *curr = Trie;
69     for (int i = 0; i < text.size(); ++i) {
70         curr = nextNode(curr, text[i] - 'a');
71         for (int j = 0; j < n; ++j)
72             if (curr->isEnd[j])
73                 cout << pattern[j] << " aparece en la posicion " << i -
74                     pattern[j].size() + 1 << '\n';
75     }
76     return 0;
77 }
```

Entrada	Salida
abcdabccabbacefdabc	abc aparece en la posicion 0
4	bcd aparece en la posicion 1
abc	abc aparece en la posicion 4
ca	ca aparece en la posicion 7
bcd	ef aparece en la posicion 13
ef	abc aparece en la posicion 16

### 3. Arreglo de sufijos.

Consideremos un string  $s$ . El arreglo de sufijos de  $s$  es un arreglo de enteros que guarda las posiciones iniciales de los sufijos de  $s$  en orden lexicográfico.

#### 3.1. Implementación

Complejidad:  $O(|s| \log |s|)$ .

```

1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4
5  #define maxn 100000 //Longitud maxima del string.
6
7  string word;          //String.
8  int n, SuffixArray[maxn]; //Arreglo de sufijos.
9
10 int rnk[maxn][2], bucket[maxn]; //Rango (SuffixArray) y Cubeta (RaxixSort).
11 int tempSA[maxn], tempRA[maxn][2]; //Arreglos temporales.
12
13 //Ordena de acuerdo a los rangos.
14 void RadixSort() {
15     int M = max(n, 256);
16     for (int k = 1; k >= 0; --k) {
17         fill_n(bucket, M, 0);
18
19         for (int i = 0; i < n; ++i)
20             ++bucket[rnk[i][k]];
21         for (int i = 1; i < M; ++i)
22             bucket[i] += bucket[i - 1];
23
24         for (int i = n - 1; i >= 0; --i) {
25             int nxt_id = --bucket[rnk[i][k]];
26             tempSA[nxt_id] = SuffixArray[i];
27             tempRA[nxt_id][0] = rnk[i][0];
28             tempRA[nxt_id][1] = rnk[i][1];
29         }
30         for (int i = 0; i < n; ++i) {
31             SuffixArray[i] = tempSA[i];
32             rnk[i][0] = tempRA[i][0];
33             rnk[i][1] = tempRA[i][1];
34         }
35     }
36 }
37
38 //Construye el arreglo de sufijos.
39 void buildSA() {
40     n = word.size();
41
42     for (int i = 0; i < n; ++i) {
43         SuffixArray[i] = i;
44         rnk[i][0] = word[i];
45     }
46     RadixSort();
47

```

```
48     for (int k = 1; k < n; k *= 2) {
49         int curr = 0, prev = rnk[0][0];
50         rnk[0][0] = curr;
51         tempSA[SuffixArray[0]] = 0;
52
53         for (int i = 1; i < n; ++i) {
54             if (rnk[i][0] != prev || rnk[i][1] != rnk[i - 1][1])
55                 ++curr;
56             prev = rnk[i][0];
57             rnk[i][0] = curr;
58             tempSA[SuffixArray[i]] = i;
59         }
60
61         for (int i = 0; i < n; ++i) {
62             int nxt_id = SuffixArray[i] + k;
63             rnk[i][1] = (nxt_id < n) ? rnk[tempSA[nxt_id]][0] : 0;
64         }
65         RadixSort();
66     }
67 }
68
69 int main() {
70     ios_base::sync_with_stdio(0); cin.tie();
71     cin >> word;
72
73     buildSA();
74     for (int i = 0; i < n; ++i) {
75         cout << SuffixArray[i] << ' ';
76         for (int j = SuffixArray[i]; j < n; ++j)
77             cout << word[j];
78         cout << '\n';
79     }
80
81     return 0;
82 }
```

Entrada	Salida
banana	5 a 3 ana 1 anana 0 banana 4 na 2 nana