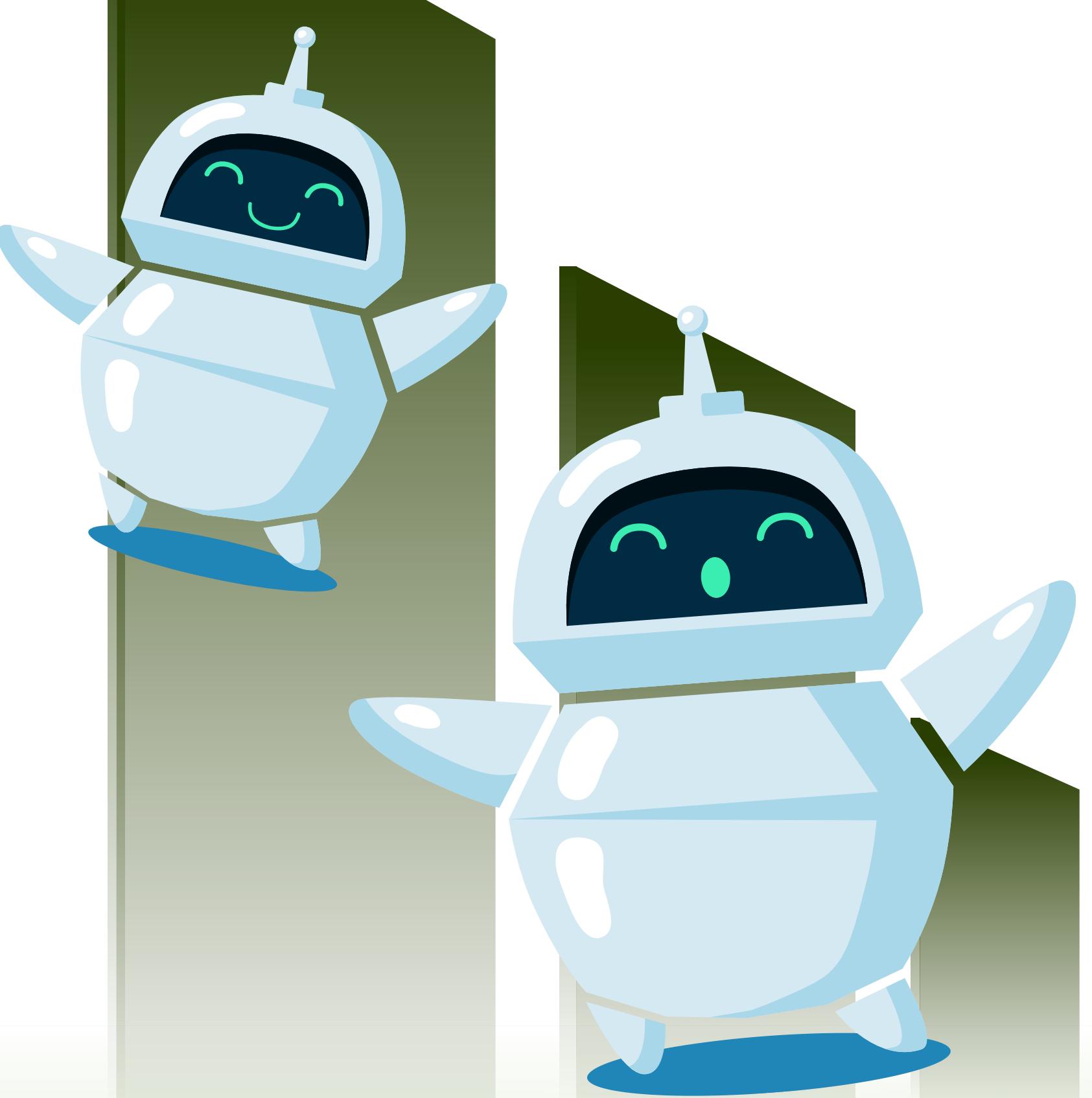


**GRUPO 5**  
**ALGORITMOS Y ESTRUCTURAS DE DATOS**

# SUFFIX- TREE



# INTRODUCCION

## BUSQUEDA DE PATRONES



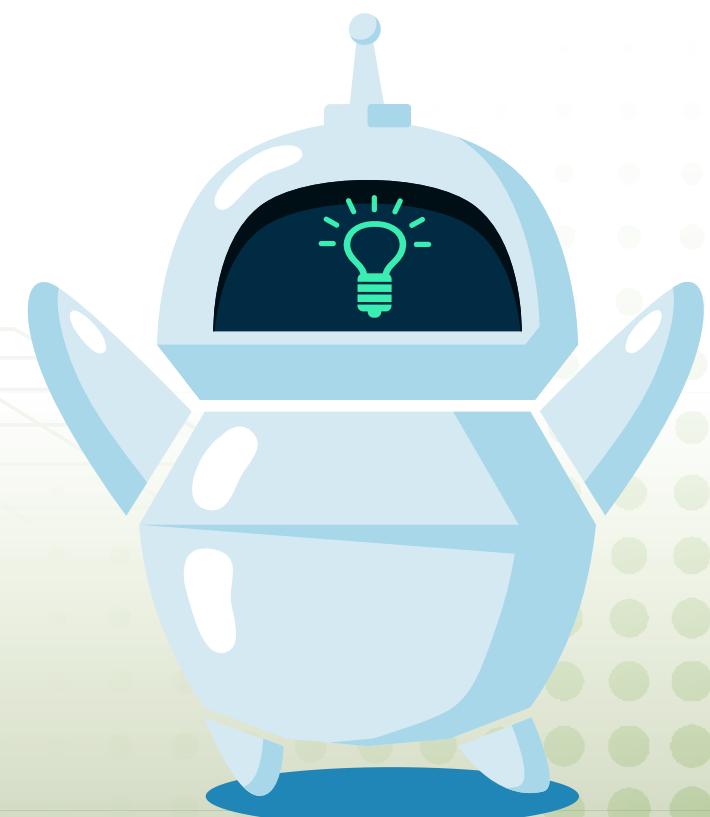
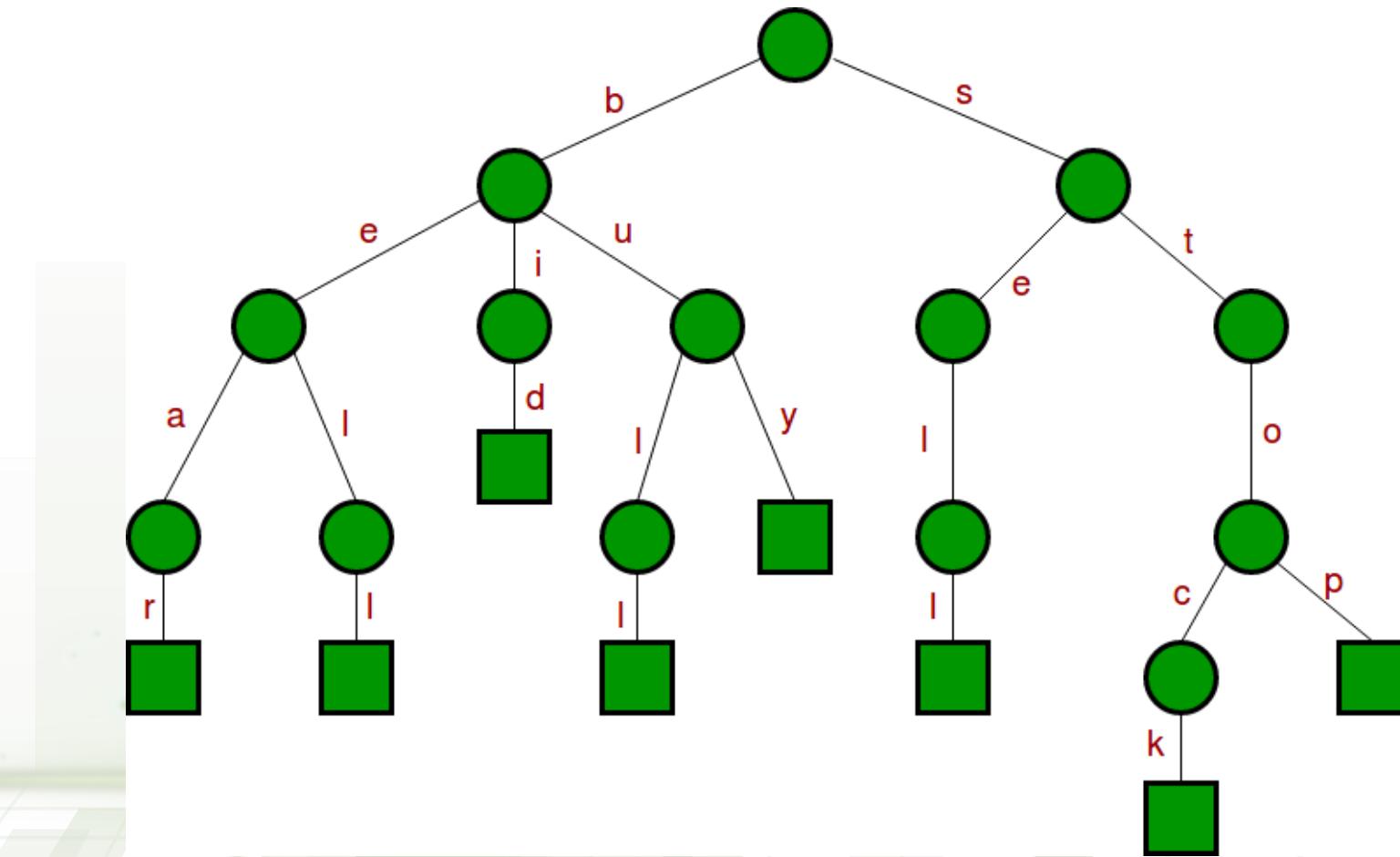
## COMPLEJIDAD METODO INGENUO

$O(nm)$ :

$n \rightarrow$  longitud del texto

$m \rightarrow$  longitud del patron buscado

## SUFFIX TREES | SUFFIX ARRAY



# CONCEPTOS PREVIOS

## SUFIJO PREFIJO

### Prefijos

Despeinar

Exalumno

Subterráneo

Telecomunicación

Ultrasónico

### Sufijos

Panadero

Bonito / Bonita

Resbaladizo

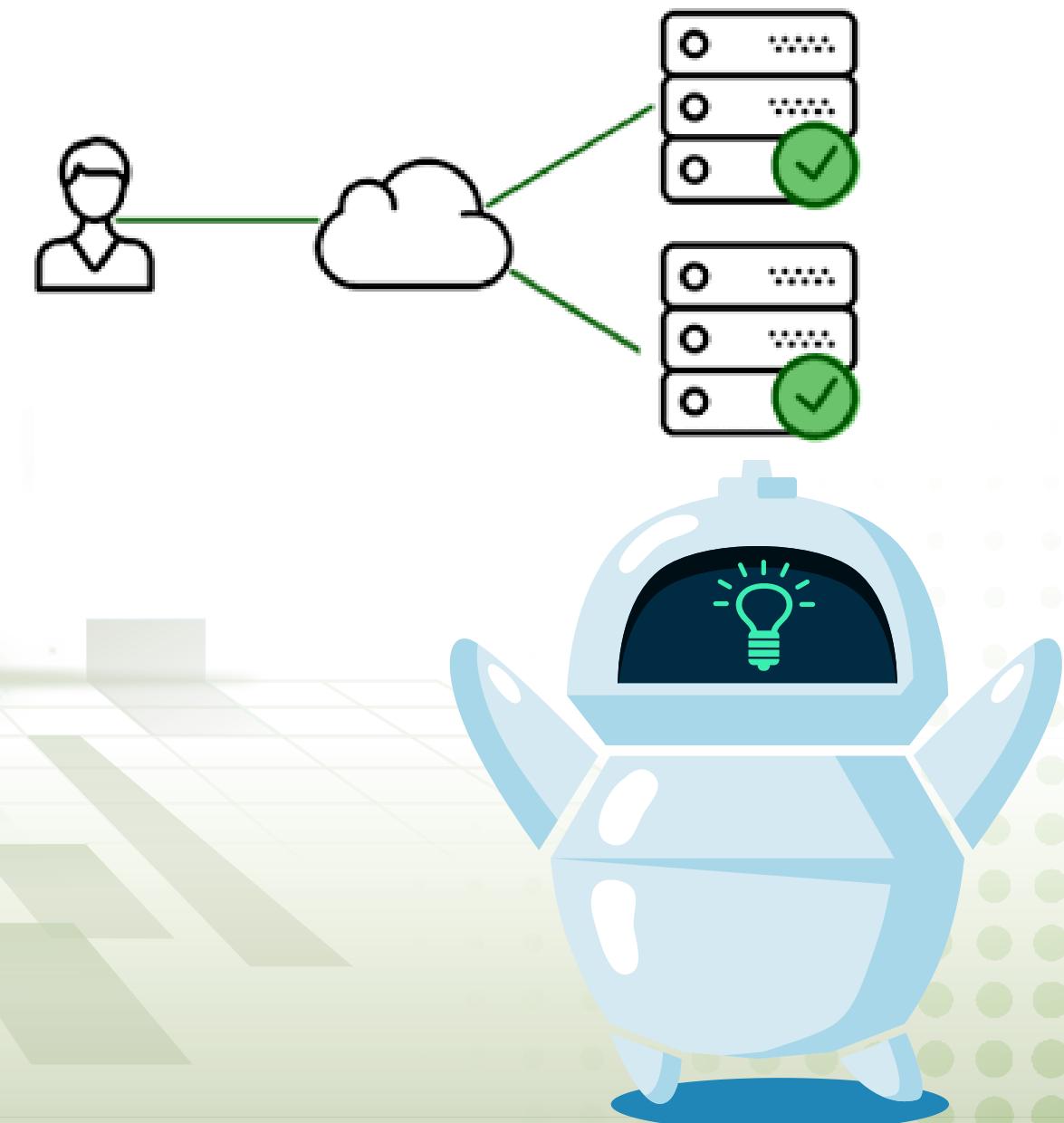
Revolucionaria

Cantamos

## ORDEN LEXICOGRAFICO

*mapa climático  
mapa económico  
mapa físico  
mapa mudo  
mapa político  
mapa tectónico  
mapa topológico*

## PROBLEMA DE BUSQUEDA DE PATRONES



# DEFINICIONES

## SUFFIX ARRAY

0	"banana\$"
1	"anana\$"
2	"nana\$"
3	"ana\$"
4	"na\$"
5	"a\$"
6	"\$"

Suffixes of String

Sorting the Suffixes  
alphabetically

6	→	"\$"
5	→	"a\$"
3	→	"ana\$"
1	→	"anana\$"
0	→	"banana\$"
4	→	"na\$"
2	→	"nana\$"

Suffix Array

## EJEMPLO BANANA

# CONSTRUCTOR

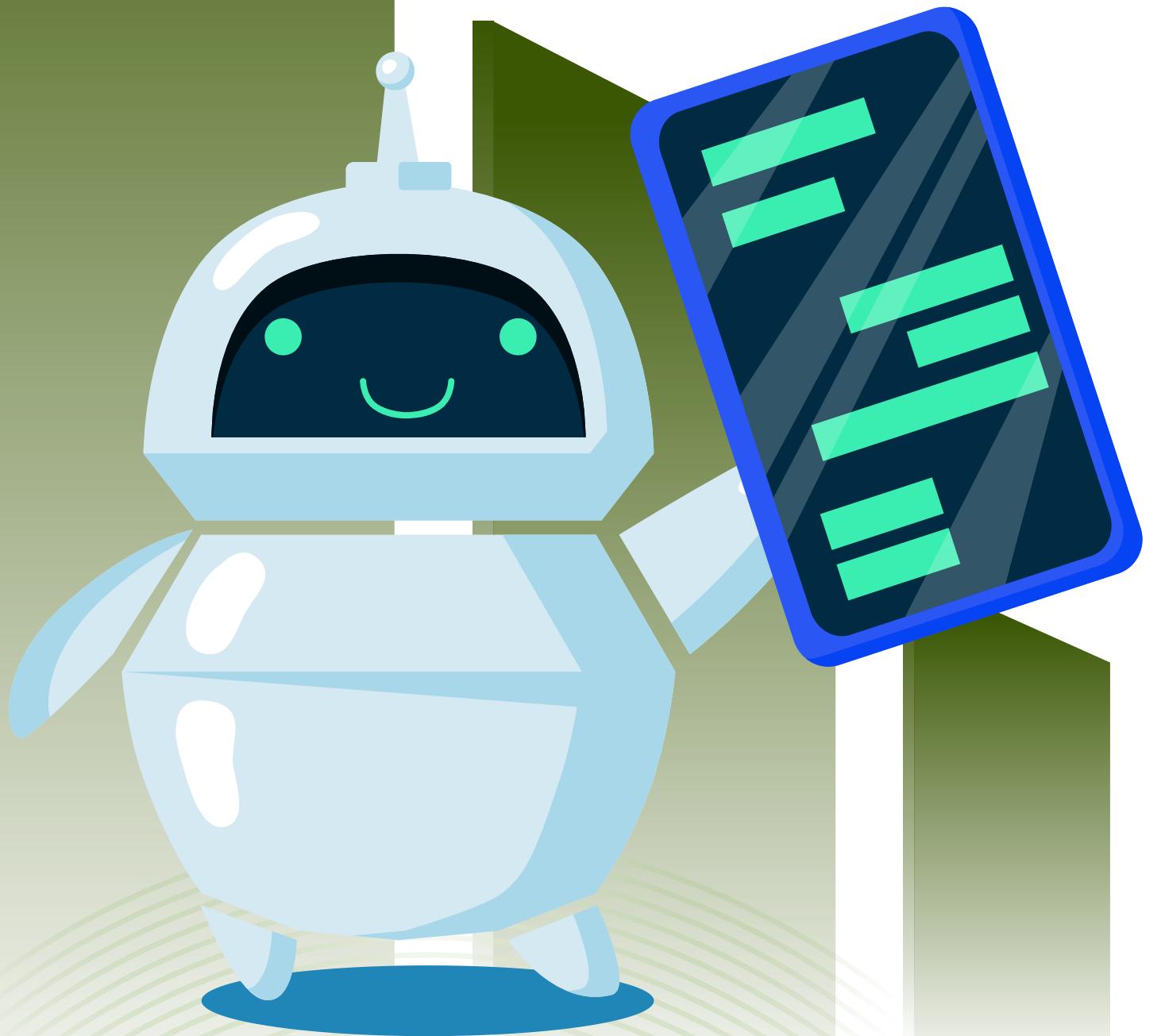
## METODOS INICIALES:

- GET TEXTO
- GET POSICION
- COMPARAR

```
8
9 class Suffix {
10    const char *texto; // puntero al texto completo
11    int posicion;    // posición donde comienza este sufijo
12
13 public:
14     // constructor
15     Suffix(const char *txt = nullptr, int pos = 0) : texto(txt), posicion(pos) {}
16
17     // getters
18     const char *getTexto() const { return texto; }
19     int getPosition() const { return posicion; }
20
21     // comparación lexicográfica entre dos sufijos
22     // retorna un valor negativo si este sufijo es menor, cero si son iguales, positivo si es mayor
23     int comparar(const Suffix &otro) const {
24         const char *s1 = texto + posicion;
25         const char *s2 = otro.texto + otro.posicion;
26         int i = 0;
27
28         while (s1[i] != '\0' && s2[i] != '\0') {
29             if (s1[i] != s2[i])
30                 return s1[i] - s2[i];
31             i++; }
32
33         // si uno es prefijo del otro, el más corto es menor
34         if (s1[i] == '\0' && s2[i] == '\0') {
35             return 0; }
36         if (s1[i] == '\0') {
37             return -1; }
38         return 1; }
```

# IMPLEMENTACIÓN

- » 🔎 MOTORES DE BÚSQUEDA Y AUTOCOMPLETADO.
- » 🧬 BIOINFORMÁTICA (GENOMAS).
- » 📊 DETECCIÓN DE PATRONES REPETIDOS.



# CONSTRUCCIÓN

Original de Manber & Myers (1993)  $\rightarrow O(n \log n)$

no se compara caracteres uno por uno, se compara por rangos  
ITERACIONES COMPARANDO LOS PRIMEROS  $2^k$  CARACTERES

$k = 0 \rightarrow 2^0 = 1 \rightarrow$  solo viendo la primera letra

i	letra	rango
0	b	1
1	a	0
2	b	1
3	a	0

$k = 1 \rightarrow 2^1 = 2 \rightarrow$  compara pares de rangos

i	sufijo	$r^{(0)}[i]$	$r^{(0)}[i+2]$	par ( $r^{(0)}[i], r^{(0)}[i+2]$ )
0	baba	1	1	(1, 1)
1	aba	0	0	(0, 0)
2	ba	1	-1	(1, -1)
3	a	0	-1	(0, -1)

par	i
(0, -1)	3
(0, 0)	1
(1, -1)	2
(1, 1)	0

$k = 2 \rightarrow 2^2 = 4 \rightarrow$

i	sufijo	$r^1[i]$	$r^1[i+4]$	par ( $r^1[i], r^1[i+4]$ )
0	baba	3	-1	(3, -1)
1	aba	1	-1	(1, -1)
2	ba	2	-1	(2, -1)
3	a	0	-1	(0, -1)

$SA = [3, 1, 2, 0]$   
que corresponde a los sufijos:  
["a", "aba", "ba", "baba"]  
 $O(n \log n)$



# BÚSQUEDA DE PATRÓN



- buscar un patrón P de longitud m con búsqueda binaria
- cada paso compara P con un sufijo  $\rightarrow O(m)$
- total:  $O(m \log n)$
- mucho más rápido que  $O(nm)$

5	3	1	0	4	2
---	---	---	---	---	---

[“a”, “ana”, “anana”, “banana”, “na”, “nana”]

buscar “na”  $\rightarrow$  mid =  $(0+5)/2 = 2$

SA[2] = 1  $\rightarrow$  “anana”

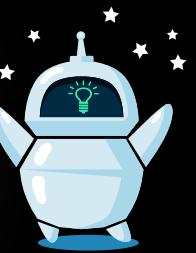
comparamos “na” con “anana”  $\rightarrow$  “na” > “anana”  
nos movemos a la derecha

mid =  $(3+5)/2 = 4$

Machine learning algorithms assist in climate  
modeling and space exploration.  
SA[4] = 4  $\rightarrow$  “na”

YEY ENCONTRADO

# VENTAJAS



## Eficiencia espacial



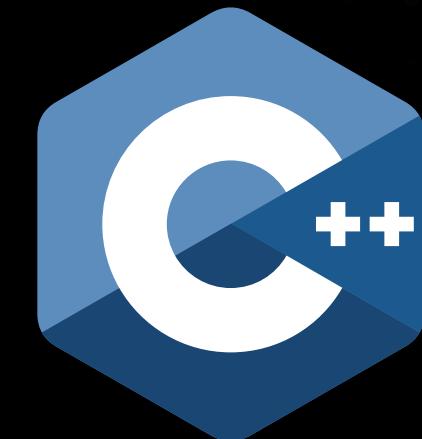
vector de enteros  
con las posiciones  
donde empiezan los  
sufijos ordenados  
memoria  
 $O(n)$



## Simplicidad de implementación



1. generar todos los sufijos del texto
2. ordenarlos lexicográficamente
3. guardar solo sus índices



## Reemplaza a Suffix Trees en casi todos los casos

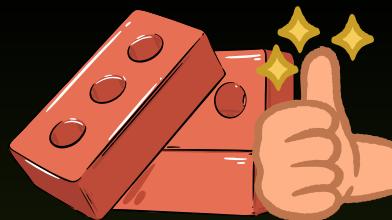
- búsqueda de subcadenas
- detección de repeticiones
- comparación de secuencias
- construcción de índices invertidos



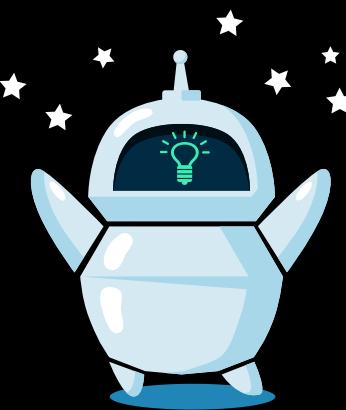
misma complejidad  
asintótica → usando  
menos memoria

## Compatibilidad con estructuras modernas

- Longest Common Prefix (LCP): indica cuánto se parecen los sufijos vecinos →
- Enhanced Suffix Arrays: similar a un Suffix Tree, pero ocupando mucha menos memoria
- Compressed Suffix Arrays y el FM-index: usan el Suffix Array como base para construir índices muy eficientes



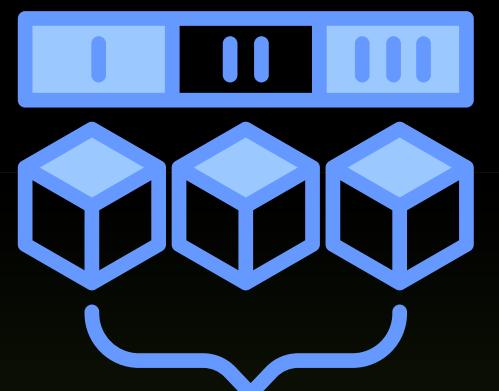
# DESVENTAJAS



Estructura estatica



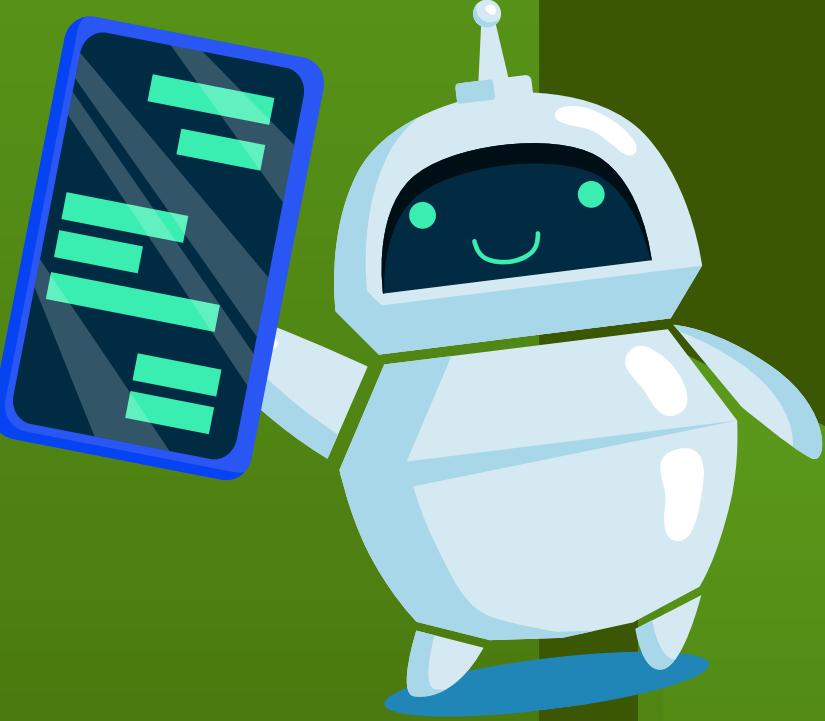
• Construcción consume memoria auxiliar.



• No permite inserciones dinámicas.



# BENEFICIOS



## PERSONALIZED RECOMMENDATIONS

Streaming services to recommend movies, music, and shows based on user preferences. Examples: E-commerce platforms utilize AI for product recommendations.

## AUTONOMOUS VEHICLES

AI enables self-driving technology in cars reducing human errors and improving road safety. Advanced AI systems process real-time data from sensors and cameras to navigate roads efficiently.

## AI IN CUSTOMER SERVICE

Chatbots and virtual agents enhance customer experience by providing 24/7 support. AI-driven sentiment analysis helps businesses understand customer feedback.

# CLASE SuffixArray

```
// clase que implementa el Suffix Array
class SuffixArray {
    const char *texto; // texto sobre el que se construye el suffix array
    int *sa;           // arreglo de sufijos
    int n;             // longitud del texto
```

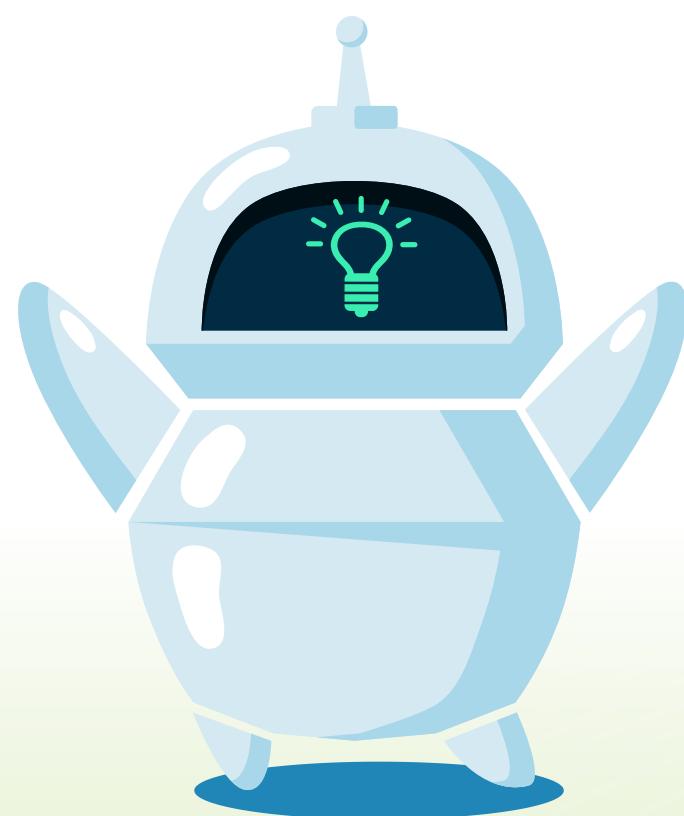


```
// constructor
SuffixArray(const char *txt) {
    texto = txt;
    n = longitudCadena(texto);
    sa = new int[n];
    construir(); }
```

# MÉTODO ESTÁTICO longitudCadena

```
// función auxiliar para obtener la longitud de una cadena
static int longitudCadena(const char *cadena) {
    int n = 0;
    while (cadena[n] != '\0')
        {n++;}
    return n;}
```

# MÉTODO construir



```
// construye el suffix array
void construir() {
    // crear arreglo de sufijos
    Suffix* sufijos = new Suffix[n];

    // inicializar sufijos
    for (int i = 0; i < n; i++) {
        sufijos[i] = Suffix(texto, i); }
```

```
// ordenar sufijos
quicksort(sufijos, inicio: 0, fin: n - 1);

// guardar las posiciones ordenadas
for (int i = 0; i < n; i++) {
    sa[i] = sufijos[i].getPosicion(); }

delete[] sufijos; }
```

# MÉTODO quicksort



```
// intercambia dos sufijos en el arreglo
static void intercambiar(Suffix *arr, int i, int j) {
    Suffix temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp; }
```

```
// partición para el quicksort
static int particion(Suffix *arr, int inicio, int fin) {
    Suffix pivot = arr[fin];
    int i = inicio - 1;
    for (int j = inicio; j <= fin - 1; j++) {
        if (arr[j].comparar(pivot) <= 0) {
            i++;
            intercambiar(arr, i, j); } }
    intercambiar(arr, i + 1, j: fin);
    return i + 1; }
```

# MÉTODO quicksort

```
// ordenamiento quicksort para los sufijos  
static void quicksort(Suffix* arr, int inicio, int fin) {  
    if (inicio < fin) {  
        int p = particion(arr, inicio, fin);  
        quicksort(arr, inicio, fin: p - 1);  
        quicksort(arr, inicio: p + 1, fin); } }
```

quicksort →  $O(n \log n)$   
generar  $n$  sufijos →  $O(n)$

ordenar los sufijos con quicksort →  $O(n \log n)$  ← complejidad de construcción

# FUNCIÓN Imprimir

```
// busca un patrón en el texto usando búsqueda binaria
// retorna la posición de la primera ocurrencia o -1 si no se encuentra
int buscar(const char* patron) const {
    int izquierda = 0;
    int derecha = n - 1;

    while (izquierda <= derecha) {
        int medio = (izquierda + derecha) / 2;
        int pos = sa[medio];

        // crear un sufijo temporal para la comparación
        Suffix sufijoTemporal(texto, pos);
        int comparacion = sufijoTemporal.compararConPatron(patron);
```

```
            if (comparacion == 0) {
                // encontrado, devolver la posición
                return pos; }
            else if (comparacion < 0) {
                izquierda = medio + 1; }
            else {
                derecha = medio - 1; } }

        return -1; } // no encontrado
```

# FUNCIÓN Buscar

```
void imprimir() const {
    cout << "suffix-array del texto: " << texto << "\n";
    cout << "índice\tsufijo\t\tposición original\n";
    for (int i = 0; i < n; i++) {
        int pos = sa[i];
        cout << i << "\t" << (texto + pos) << "\t\t" << pos << "\n"; } }
```

# Casos de prueba

Suffix Array del texto: banana

Indice	Sufijo	Posicion original
0	a	5
1	ana	3
2	anana	1
3	banana	0
4	na	4
5	nana	2

Patron encontrado en posicion: 1

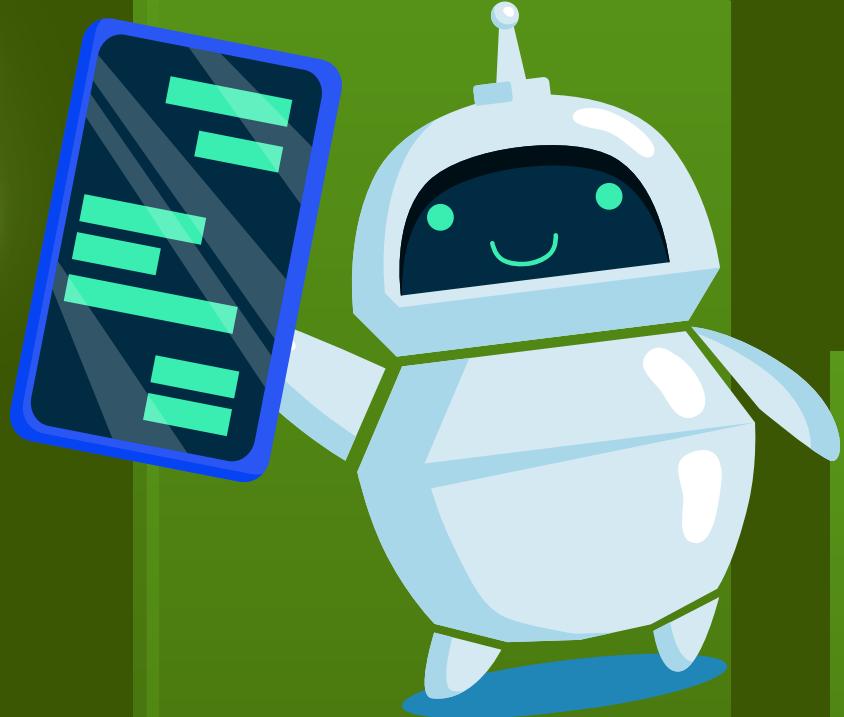
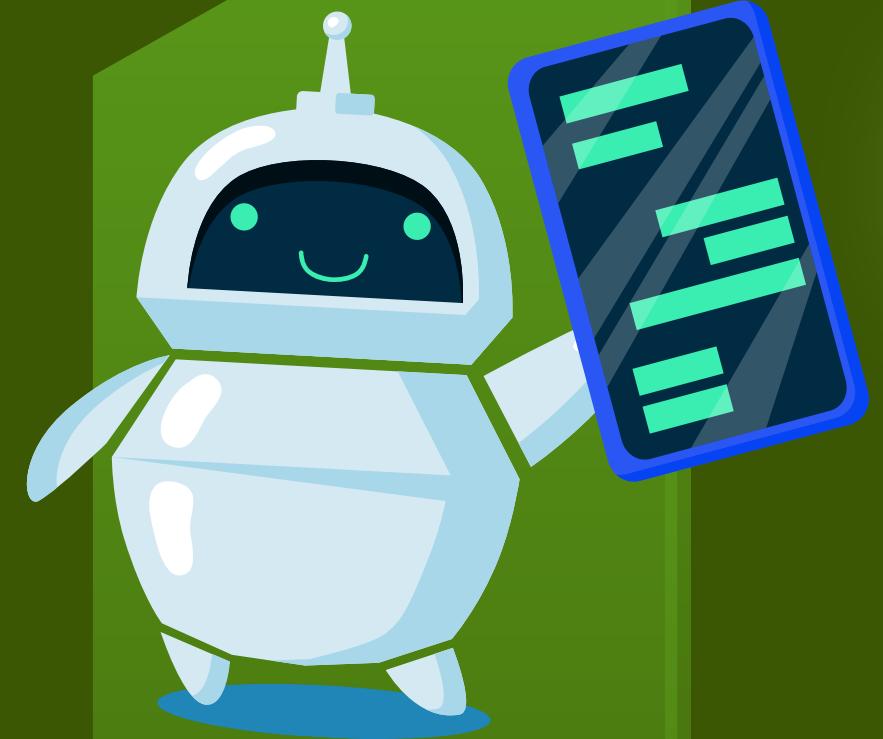
Suffix Array del texto: banana

Indice	Sufijo	Posicion original
0	a	5
1	ana	3
2	anana	1
3	banana	0
4	na	4
5	nana	2

Patron encontrado en posicion: 0

# CONCLUSIONES

- El Suffix Array permite búsquedas eficientes con complejidad  $O(m \log n)$ .
- Su principal debilidad es ser una estructura estática.
- Es base de herramientas modernas como compresión y búsqueda genómica.





# THANK YOU!

[www.reallygreatsite.com](http://www.reallygreatsite.com)

@reallygreatsite