Tecnológico de Monterrey.

Campus Querétaro.

TC2038. Análisis y diseño de Algoritmos A

M.C. Ramona Fuentes Valdéz

rfuentes@tec.mx

Técnicas de diseño de algoritmos

Manejo de strings

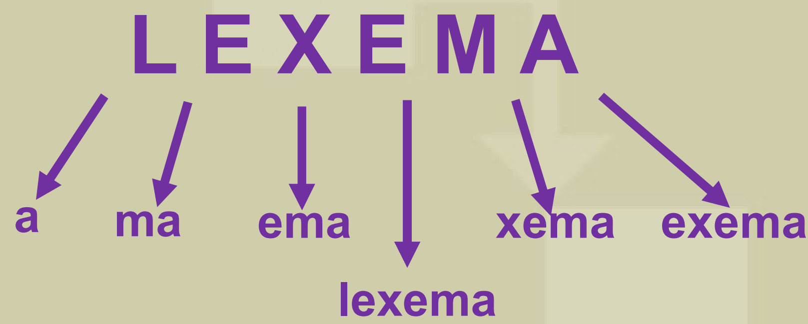
Arreglos de sufijos (Suffix Array)

- El Arreglo de sufijos (*Suffix Array*) es una estructura de datos que fue propuesta por Udi Manber y Gene Myers en 1990.
- Es muy utilizada en varias aplicaciones relacionadas con el análisis de cadenas.
- Se define de la siguiente forma:
  - Dada una cadena *S*, de longitud *n*, su arreglo de sufijos es un arreglo de enteros que contiene las posiciones que tienen los *n + 1* sufijos en la cadena T = S$, ordenados lexicográficamente, considerando $ como el primer carácter del alfabeto.

Sufijos…

- Los sufijos son morfemas (*unidades mínimas de significado*) que se colocan al final de una palabra o lexema y le aportan algún matiz de sentido o información gramatical.

  Ejemplo: *galanc**ete**, caser**ío**, trist**ísimo***.



El sufijo *j* de *x=[1… n]* es la subcadena *x[j…n]*.

Técnicas de diseño de algoritmos

Manejo de strings

Arreglos de sufijos (Suffix Array)

- La matriz de sufijos SA de *x* es una matriz de *n* números enteros tal que:
  - *SA[i]=j*, el sufijo *j* de *x* tiene rango *i* en el orden lexicográfico de todos los sufijos de *x*.

Ejemplo:

- Para *s* = abaab

Los sufijos son:	→ Después de ordenar:	→ El arreglo de sufijos de *s* será:
0. abaab	2. aab	{2, 3, 0, 4, 1}
1. baab	3. ab	
2. aab	0. abaab	
3. ab	4. b	
4. b	1. baab	

```
Mississippi     1: mississippi   11: i
                2: ississippi     8: ippi
                3: ssissippi      5: issippi
                4: sissippi       2: ississppi
                5: issippi        1: mississippi
                6: ssippi        10: pi
                7: sippi          9: ppi
                8: ippi           7: sippi
                9: ppi            4: sissippi
               10: pi             6: ssippi
               11: i              3: ssissippi
```

$SA = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$

Técnicas de diseño de algoritmos

Manejo de strings

Arreglos de sufijos (Suffix Array)

Aplicaciones

- Un arreglo de sufijos es una estructura de datos extremadamente útil, se puede utilizar para una amplia gama de problemas, algunos como:
  - Búsqueda de patrones (especialmente útil en Bioinformática)
    - Se puede utilizar como índice para localizar rápidamente cada aparición de un patrón de subcadena P dentro de la cadena S.
  - Encontrar la subcadena repetida más larga
    - Encontrar cada aparición del patrón equivale a encontrar cada sufijo que comienza con la subcadena.
  - Encontrar la subcadena común más larga
    - Con el orden lexicográfico, estos sufijos se agruparán en la matriz de sufijos y se podrán encontrar de manera eficiente con dos búsquedas binarias. La primera búsqueda localiza la posición inicial del intervalo y la segunda determina la posición final.
  - Encontrar el palíndromo más largo de una cadena.
  - Bibliometría. Métodos estadísticos para análisis de libros, entre otros.
  - Algoritmos. Es usado en otros algoritmos como de compresión.
  - Patrones. Analizador de cadenas de texto.

Técnicas de diseño de algoritmos

Manejo de strings

Arreglos de sufijos (Suffix Array)

Algoritmo

//SUFFIX_ARRAY
1. Formar la cadena de trabajo T ← S$
2. Formar todos los sufijos de T
3. Ordenar lexicográficamente los sufijos encontrados considerando a $ como primer símbolo del alfabeto
4. Formar el arreglo A con las posiciones de inicio en las cadenas T de cada uno de los sufijos

return A

Ejemplo:

- Para el *string* “banana$”

0	1	2	3	4	5	6
b	a	n	a	n	a	$

i - index	sufijo
0	banana
1	anana
2	nana
3	ana
4	na
5	a
6	$

Se ordenan los sufijos alfabéticamente

i - index	sufijo
6	$
5	a
3	ana
1	anana
0	banana
4	na
2	nana

Suffix Array:
{6, 5, 3, 1, 0, 4, 2}

Técnicas de diseño de algoritmos

Manejo de strings

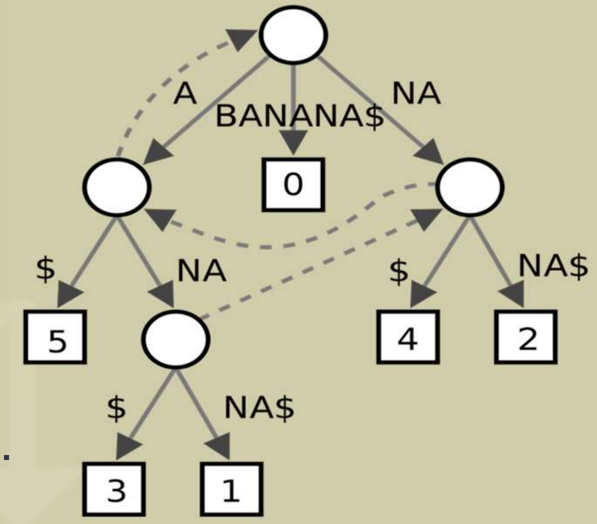
Arreglos de sufijos (Suffix Array)

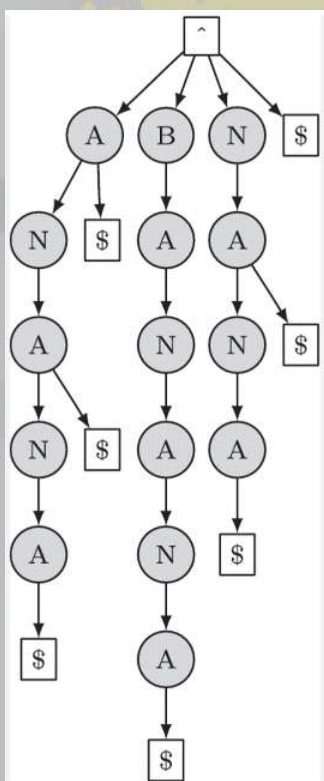
Árbol de sufijos

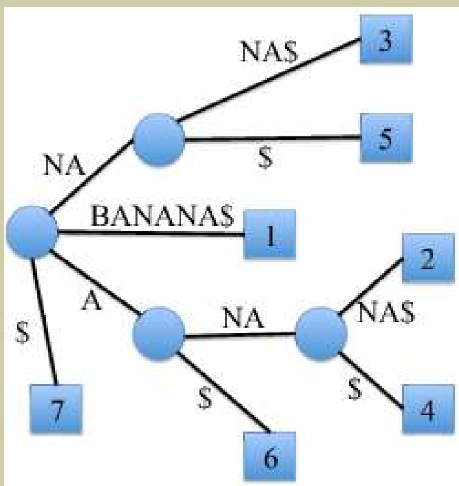
- Este es un árbol que representa de manera gráfica todos los sufijos que pueda tener un *String* S, dado.

Ejemplo:

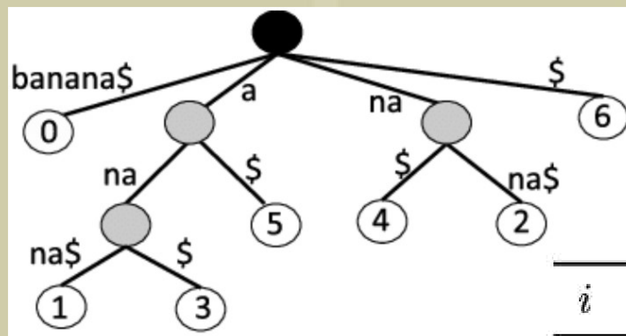
- Para el *string* “BANANA$” donde ($ denota fin de cadena).







0	1	2	3	4	5	6
b	a	n	a	n	a	$



- Las etiquetas de borde en una ruta desde la raíz hasta una hoja corresponden a un sufijo en *S*.
- Las etiquetas de hoja muestran el desplazamiento de cada sufijo en *S*.

i	Suffix	Sorted Suffix	SA[i]
0	banana$	$	6
1	anana$	a$	5
2	nana$	ana$	3
3	ana$	anana$	1
4	na$	banana$	0
5	a$	na$	4
6	$	nana$	2

Técnicas de diseño de algoritmos

Manejo de strings

Arreglos de sufijos (Suffix Array)

Construcción del arreglo de sufijos

- Una forma ingenua (*naïve*) de crear la matriz de sufijos sería almacenar todos los sufijos en una matriz y ordenarlos.
- Si utilizamos un algoritmo de ordenamiento basado en comparación O(N log (N)), entonces el tiempo total para crear la matriz de sufijos sería O(N^2logN), porque la comparación de cadenas requiere un tiempo O(N).
- Esto es demasiado lento para cadenas grandes.

Sobre el desarrollo de este algoritmo…

A Taxonomy of Suffix Array Construction Algorithms

Documento de ACM:

Puglisi, S. J., Smyth,W. F., and Turpin, A. H. 2007. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.* 39, 2, Article 4 (June 2007), 31 pages DOI = 10.1145/1242471.1242472 http://doi.acm.org/10.1145/124247

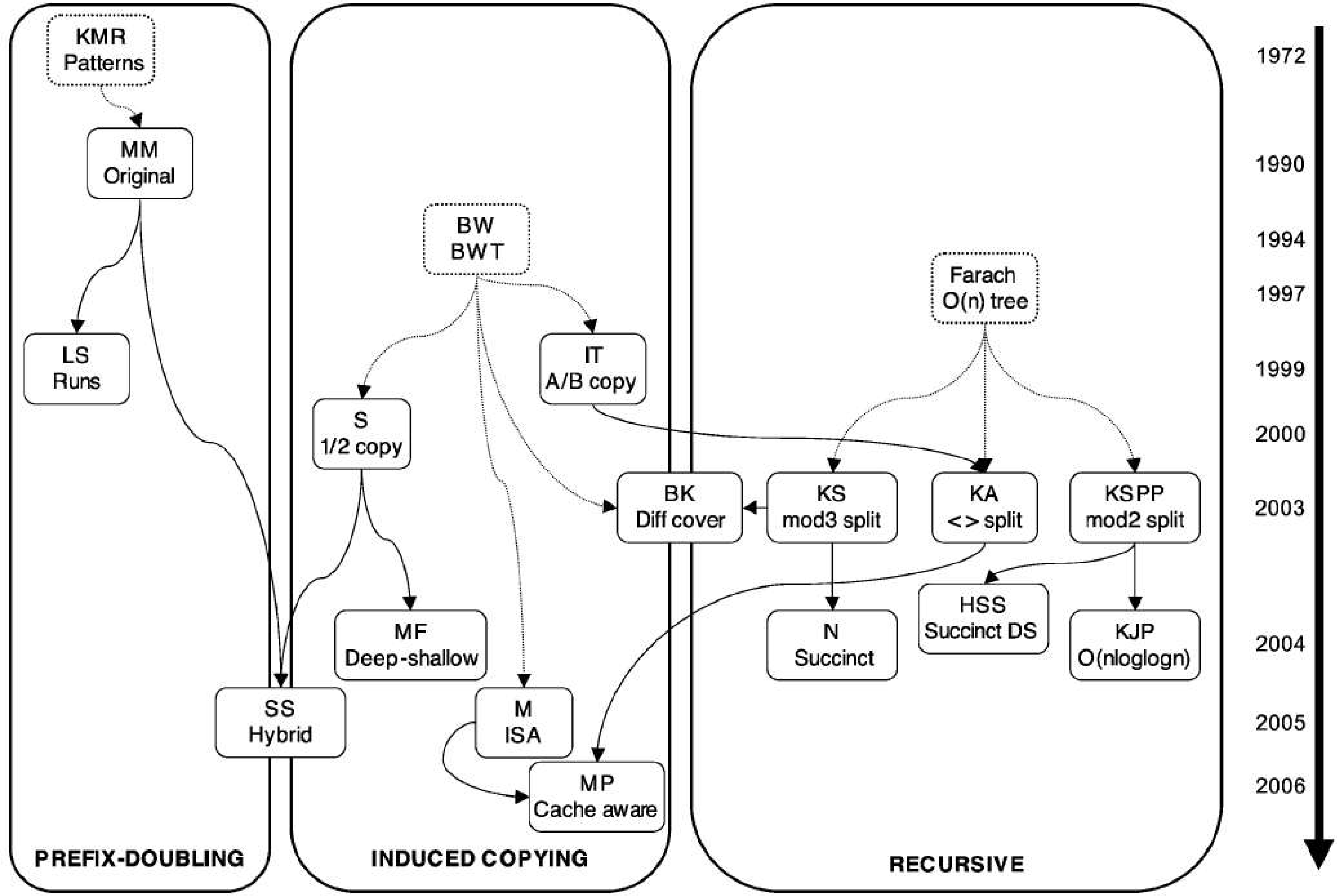
- En este artículo se enumeran los tiempos de funcionamiento de 17 SACAs. Hoy en día, la construcción original propuesta por Manber y Myers es unas 30 veces más lenta que la SACA más rápida conocida hasta ahora. El camino sobre este tema aún no ha terminado, se están desarrollando nuevos algoritmos e implementaciones.
- Una idea es la de duplicar prefijos (*prefix doubling*). Es el fundamento del algoritmo MM original (1990). Una versión modificada de Larsson y Sadakane (1999) es “sólo” un factor 3 más lenta que la mejor versión actual.

Técnicas de diseño de algoritmos

Manejo de strings

Arreglos de sufijos (Suffix Array)

A Taxonomy of Suffix Array Construction Algorithms



Documento de ACM: **Fig. 2.** Taxonomy of suffix array construction algorithms.

Puglisi, S. J., Smyth,W. F., and Turpin, A. H. 2007. A taxonomy of suffix array construction algorithms. *ACM*

Técnicas de diseño de algoritmos

Manejo de strings

Arreglos de sufijos (Suffix Array)

A Taxonomy of Suffix Array Construction Algorithms

Table I. Performance Summary of the Construction Algorithms

Algorithm	Worst Case	Time	Memory
Prefix-Doubling			
MM [Manber and Myers 1993]	$O(n \log n)$	30	$8n$
LS [Larsson and Sadakane 1999]	$O(n \log n)$	3	$8n$
Recursive			
KA [Ko and Aluru 2003]	$O(n)$	2.5	7–10n
KS [Kärkkäinen and Sanders 2003]	$O(n)$	4.7	10–13n
KSPP [Kim et al. 2003]	$O(n)$	—	—
HSS [Hon et al. 2003]	$O(n)$	—	—
KJP [Kim et al. 2004]	$O(n \log\log n)$	3.5	13–16n
N [Na 2005]	$O(n)$	—	—
Induced Copying			
IT [Itoh and Tanaka 1999]	$O(n^2 \log n)$	6.5	$5n$
S [Seward 2000]	$O(n^2 \log n)$	3.5	$5n$
BK [Burkhardt and Kärkkäinen 2003]	$O(n \log n)$	3.5	5–6n
MF [Manzini and Ferragina 2004]	$O(n^2 \log n)$	1.7	$5n$
SS [Schürmann and Stoye 2005]	$O(n^2)$	1.8	9–10n
BB [Baron and Bresler 2005]	$O(n\sqrt{\log n})$	2.1	$18n$
M [Maniscalco and Puglisi 2007]	$O(n^2 \log n)$	1.3	5–6n
MP [Maniscalco and Puglisi 2006]	$O(n^2 \log n)$	1	5–6n
Hybrid			
IT+KA	$O(n^2 \log n)$	4.8	$5n$
BK+IT+KA	$O(n \log n)$	2.3	5–6n
BK+S	$O(n \log n)$	2.8	5–6n
Suffix Tree			
K [Kurtz 1999]	$O(n \log \sigma)$	6.3	13–15n

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].

Documento de ACM:

Puglisi, S. J., Smyth,W. F., and Turpin, A. H. 2007. A taxonomy of suffix array construction algorithms. *ACM*

Preguntas

Técnicas de diseño de algoritmos

Manejo de strings

Subcadenas

- Subsecuencia creciente más larga (*Longest Increasing Subsequence* - LIS)
- Subsecuencia común más larga (*Longest Common Subsequence* - LCS)
- Subcadena común más larga (*Longest Common Substring*)

Subsecuencia

- Considera una cadena A = [a, b, c, d].
  - Una subsecuencia (también llamada subcadena) se obtiene eliminando 0 o más símbolos (*no necesariamente consecutivos*) de A.

    Ejemplo:

    - Subsecuencias de A: [a, b, c, d] , [a], [b], [c], [d], [a, d], [a, c], [b, d].
    - No son subsecuencias de A: [d, b], [d, a], [d, a, c].

Técnicas de diseño de algoritmos

Manejo de strings

Subsecuencia creciente más larga (*Longest Increasing Subsequence* - LIS)

- Dado un arreglo de *n* símbolos comparables, dar la subsecuencia creciente más larga.

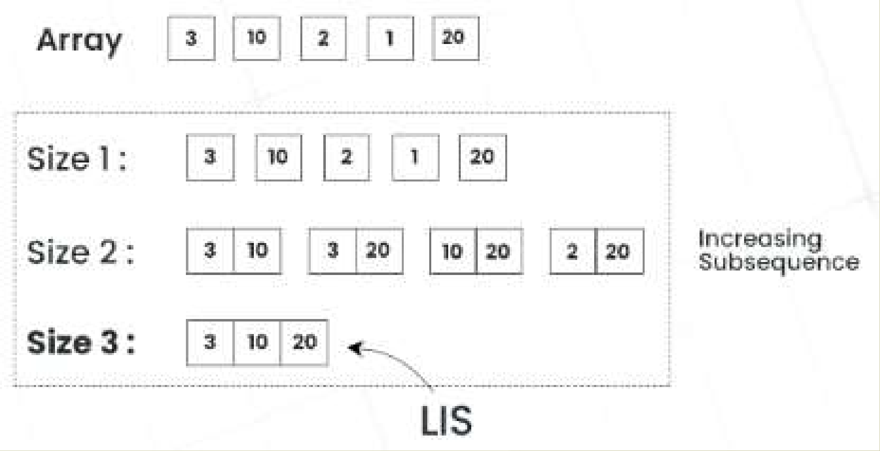
  Ejemplo:
  - Para A = [-7, 10, 9, 2, 3, 8, 8, 1]

    → la subsecuencia creciente más larga es [-7, 2, 3, 8].

```
// Algoritmo LIS
Input: A : Array
  Aux : Array
  Aux[1] ← 1
  for i ← 2 to n do
    count ← 0
    for j ← 1 to (i − 1) do
      if A[j] < A[i] then
        count ← MAX(count, Aux[j])
      end if
    end for
    Aux[i] ← count + 1
  end for

  count ← 0
  for i ← 1 to n do
    count ← MAX(count, Aux[i])
  end for
return count
```

Input Sequence	**6**	**9**	**8**	**2**	**3**	**5**	**1**	**4**	**7**
LIS 1	**2**	**3**	**4**	**7**					
LIS 2	**2**	**3**	**5**	**7**					



Técnicas de diseño de algoritmos

Manejo de strings

Subsecuencia común más larga (*Longest Common Subsequence* - LCS)

- Si a una secuencia *S* de elementos le quitamos algunos de ellos y dejamos los que quedan en el orden en el que aparecían originalmente tenemos lo que se llama una *subsecuencia* de *S*.

  Ejemplo:

  - Si A = “Matamoscas”

    → ”aaoa” es una subsecuencia de la secuencia ”Matamoscas”.

- El término también se aplica cuando se quitan todos los elementos (*es decir la secuencia vacía es siempre subsecuencia de cualquier secuencia*) o cuando no quitamos ninguno (*lo que significa que cualquier secuencia es siempre subsecuencia de sí misma*).

Ejercicio:

- Dadas dos cadenas, A = [c, d, c, c, f, g, e] y B = [e, c, c, e, g, f, e], determinar la subsecuencia común más larga de ambas cadenas.

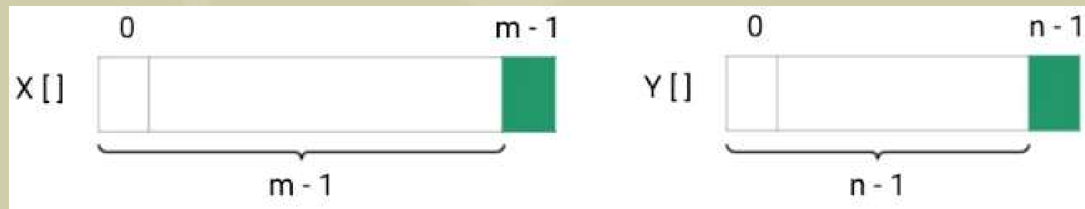
Técnicas de diseño de algoritmos

Manejo de strings

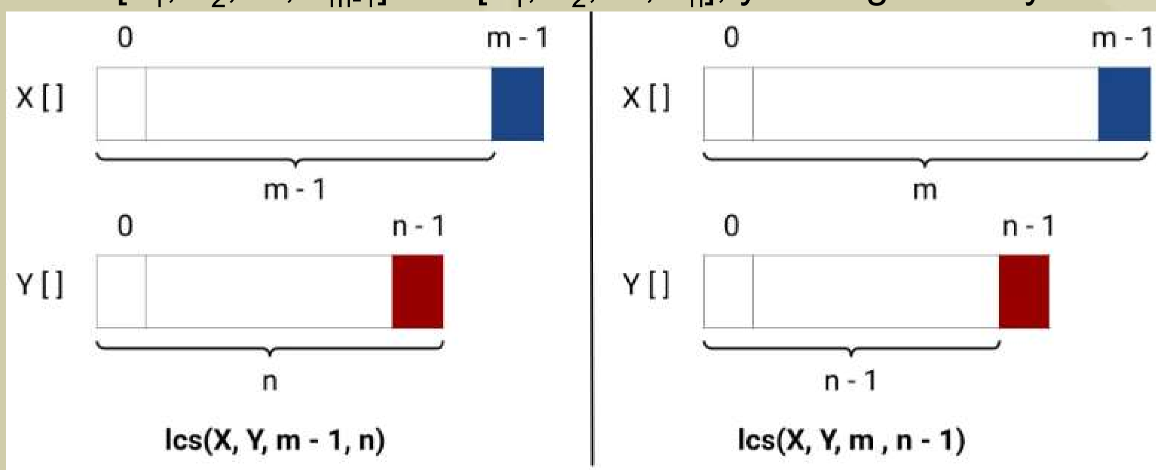
Subsecuencia común más larga (*Longest Common Subsequence* - LCS)

- Considere dos secuencias A = [a_1, a_2, ..., a_m] y B = [b_1, b_2, ..., b_n]
- Para resolver el problema hay que revisar los últimos dos símbolos: a_m y b_n..
- Se observa que hay dos posibilidades:
  - Caso 1: $a_m = b_n$. En este caso, la subsecuencia común más larga debe contener a_m. Por lo que basta con encontrar la subsecuencia común más larga de [a_1, a_2, ..., a_{m-1}] y [b_1, b_2, ..., b_{n-1}].

```
if ( X[m - 1] == Y[n - 1] )
    lcs( X, Y, m, n ) = 1 + lcs( X, Y, m - 1, n - 1 )
```



  - Caso 2: $a_m \neq b_n$. En este caso, puede hacerse corresponder [a_1, a_2, ..., a_m] con [b_1, b_2, ..., b_{n-1}] y también [a_1, a_2, ..., a_{m-1}] con [b_1, b_2, ..., b_n], y se elige el mayor de los dos resultados.



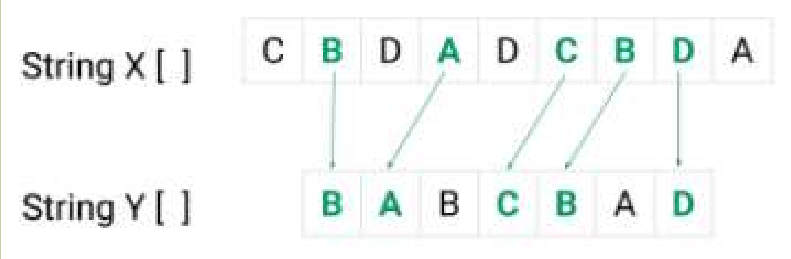

```
if ( X[m-1] != Y [n-1] )
    lcs(X, Y, m, n) = max ( lcs(X, Y, m - 1 , n), lcs(X, Y, m , n - 1) )
```

Técnicas de diseño de algoritmos

Manejo de strings

Subsecuencia común más larga (*Longest Common Subsequence* - LCS)

- Si a una secuencia *S* de elementos le quitamos algunos de ellos y dejamos los que quedan en el orden en el que aparecían originalmente tenemos lo que se llama una *subsecuencia* de *S*.



Longest common subsequence is: B A C B D
So the longest length = 5

Algoritmo

```
// Algoritmo LCS
Input: A,B : Array
   M : Matrix[0..A.length][0..B.length]
   m ← A.length
   n ← B.length
   INIT (M, 0)
      for i ← 1 to m do
         for j ← 1 to n do
            if A[i] = B[j] then
               M[i][j] ← 1 + M[i − 1][j − 1]
            else
               M[i][j] ← MAX(M[i − 1][j], M[i ][j − 1])
            end if
         end for
      end for
      return M[m][n]
```

```
String A = "acbaed";
String B = "abcadf";
```

String A	a	c	b	a	e	d
String B	a	b	c	a	d	f

Longest Common
Subsequence(LCS): acad
Length: 4

Técnicas de diseño de algoritmos

Manejo de strings

Subcadena común más larga (*Longest Common Substring*)

- El problema consiste en encontrar aquella subcadena **continua** que puede coincidir en todas las cadenas de entrada no necesariamente del mismo tamaño.
- Este problema puede arrojar múltiples soluciones.

  Ejemplo:
  - Para S1 = “AABCABA” y S2 = “CABCBABACC”
    → el tamaño de la subcadena más larga es 3: “ABC”, “CAB” y “ABA”.
- Considere dos secuencias $A = [a_1, a_2, ..., a_m]$ y $B = [b_1, b_2, ..., b_n]$
- Para resolver el problema hay que revisar los últimos dos símbolos: a_m y b_n..
- Se observa que hay dos posibilidades:
  - Caso 1: $a_m = b_n$. En este caso, la subcadena común más larga debe contener a_m. Primero se debe encontrar la subcadena común más larga de $[a_1, a_2, ..., a_{m-1}]$ y $[b_1, b_2, ..., b_{n-1}] + 1$, y luego comparar ese resultado con el máximo previo.
  - Caso 2: $a_m \neq b_n$. En este caso, la longitud más larga será cero.

Técnicas de diseño de algoritmos

Manejo de strings

Subcadena común más larga (*Longest Common Substring*)

- El problema consiste en encontrar aquella subcadena **continua** que puede coincidir en todas las cadenas de entrada no necesariamente del mismo tamaño.
- Este problema puede arrojar múltiples soluciones.

```
//Algoritmo LCS
Input: A,B : Array
  M : Matrix[0..A.length][0..B.length]
  m ← A.length
  n ← B.length
  INIT(M, 0)
  for i ← 1 to m do
    for j ← 1 to n do
      if A[i] = B[j] then
        M[i][j] ← 1 + M[i − 1][j − 1]
        maximum ← MAX(M[i][j], max)
      else
        M[i][j] ← 0
      end if
    end for
  end for
  return maximum
```

	S	I	C	K
B	0	0	0	0
R	0	0	0	0
I	0	0	0	0
C	0	0	0	0
K	0	0	0	0
S	0	0	0	0

	S	I	C	K
B	0	0	0	0
R	0	0	0	0
I	0	1	0	0
C	0	0	0	0
K	0	0	0	0
S	0	0	0	0

	S	I	C	K
B	0	0	0	0
R	0	0	0	0
I	0	1	0	0
C	0	0	2	0
K	0	0	0	3
S	1	0	0	0

$$C[i, j] = \begin{cases} 0 & \text{If } i == 0 \text{ or } j == 0 \\ 1 + C[i-1, j-1] & \text{If } i, j > 0 \text{ and } S1[i] == S2[j] \\ 0 & \text{If } i, j > 0 \text{ and } S1[i] \mathrel{!=} S2[j] \end{cases}$$

InterviewBit

Técnicas de diseño de algoritmos

Manejo de strings

Subcadena común más larga (*Longest Common Substring*)

- El problema consiste en encontrar aquella subcadena **continua** que puede coincidir en todas las cadenas de entrada no necesariamente del mismo tamaño.
- Este problema puede arrojar múltiples soluciones.

		A	B	C	X	Y	Z	A	Y
	0	0	0	0	0	0	0	0	0
X	0	0	0	0	1	0	0	0	0
Y	0	0	0	0	0	2	0	0	1
Z	0	0	0	0	0	0	3	0	0
A	0	1	0	0	0	0	0	4	0
B	0	0	2	0	0	0	0	0	0
C	0	0	0	3	0	0	0	0	0
B	0	0	1	0	0	0	0	0	0

InterviewBit