Tecnológico de Monterrey.

Campus Querétaro.

TC2038. Análisis y diseño de Algoritmos A

M.C. Ramona Fuentes Valdéz

rfuentes@tec.mx

1

Técnicas de diseño de algoritmos

Manejo de strings

Introducción

- El manejo de texto involucra una gran cantidad de información, por lo que es evidente la necesidad de automatizar ciertas tareas del manejo de texto.

El problema…

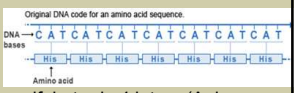
- Dado un texto T y un patrón P, verifique si P ocurre en T
  - Ejemplo:
    - Si T = {aabbcbbcabbbcbccccabbabbccc}
      - Encuentre todas las apariciones del patrón P = bbc
- Hay variaciones de combinación de patrones.
  - Encontrar coincidencias "aproximadas"
  - Encontrar múltiples patrones, etc.

2

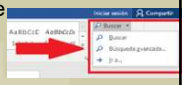
Técnicas de diseño de algoritmos

Manejo de strings

Aplicaciones



- Biología computacional
  - La secuencia de ADN es una palabra larga *(o texto)* sobre un alfabeto de 4 letras (A de adenina, C de citosina, G de guanina y T de tiamina),
  - GTTTGAGTGGTCAGTCTTTTCGTTTCGACCAGACCTCAGTTCGCTTAGAGCAGCCGAAA…
  - Encuentra un patrón específico W
- Encontrar patrones en documentos formados con un alfabeto grande
  - Procesamiento de textos
  - Búsqueda web



- Detección de errores ortográficos

- Cadenas coincidentes de bytes que contienen
  - Datos gráficos
  - Código máquina

```
0000: CA FE BA BE 00 00 00 34 00 14 0A 00 04 00 10 09
...
...
...
0100: 00 01 00 09 00 00 00 39 00 02 00 01 00 00 00 0B
0110: 2A B7 00 01 2A 10 2A B5 00 02 B1 00 00 00 02 00
0120: 0A 00 00 00 0A 00 02 00 00 00 03 00 04 00 04 00
0130: 0B 00 00 00 0C 00 01 00 00 00 0B 00 0C 00 0D 00
0140: 00 00 01 00 0E 00 00 00 02 00 0F
```

- grep en unix
  - grep busca líneas que coincidan con un patrón.
- Entre otros…

```
test@test-VirtualBox: ~/Desktop/files
File  Edit  View  Search  Terminal  Help
test@test-VirtualBox:~/Desktop/files$ grep "phoenix number3" *
sample3:phoenix number3
sample3:phoenix number3 test2
sample3:phoenix number3 test3
test@test-VirtualBox:~/Desktop/files$ grep -x "phoenix number3" *
sample3:phoenix number3
test@test-VirtualBox:~/Desktop/files$
```


3

Técnicas de diseño de algoritmos

Manejo de strings

Conceptos básicos

- Una cadena de caracteres (*string*) es una secuencia ordenada de caracteres.
  - El primer carácter en la cadena es el que se encuentra más a la izquierda y ocupa la posición 1, el siguiente la posición 2 y así sucesivamente.
  - La longitud de una cadena *S*, |*S*|, está dada por el número de caracteres que contiene.
- Una subcadena (*substring*) de la cadena *S* es cualquier cadena de caracteres que se encuentran en *S*.
  - La subcadena S[i ..j] es la cadena formada por los caracteres de *S* que están de la posición *i* a la *j*, inclusive.
    - Prefijo: dada una cadena *S*, un prefijo *S[1..i]* del mismo es cualquier subcadena que inicia en la posición 1 y termina en la posición i.
    - Sufijo: dada una cadena *S*, un sufijo *S[i ..n]* del mismo es cualquier subcadena que inicia en la posición i y termina en el último elemento de *S*.
- Trabajar con cadenas suele ir de la mano con las estructuras de datos.
  - Ejemplo: Mapas hash.
    - Para contar el número de ocurrencias de una determinada palabra en un texto.
      - Un mapa se puede utilizar con la palabra como llave y las ocurrencias como el valor.

4

Técnicas de diseño de algoritmos

Manejo de strings

Naive

- Es el método más simple que utiliza el enfoque de fuerza bruta.
- Es un enfoque sencillo para resolver el problema.
- Compara el primer carácter del patrón con el texto que se puede buscar.
  - Si se encuentra una coincidencia, se avanzan los punteros en ambas cadenas.
  - Si no se encuentra la coincidencia, el puntero del texto se incrementa y el puntero del patrón se reestablece.
  - Este proceso se repite hasta el final del texto.
- No requiere ningún procesamiento previo. Comienza directamente a comparar ambas cadenas carácter por carácter.
- Complejidad O(*m*(n-m)*)

5

Técnicas de diseño de algoritmos

Manejo de strings

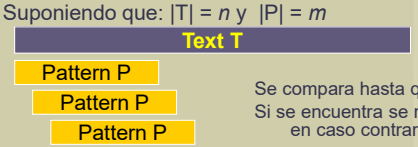
Naive

Algoritmo:

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

Ejemplo:

- Suponiendo que: |T| = *n* y |P| = *m*



Se compara hasta que se encuentra una coincidencia.
Si se encuentra se regresa el índice en donde ocurre,
en caso contrario se regresa un -1.

Otros ejemplos:

Input: *txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"*
Output: *Pattern found at index 10*

Input: *txt[] = "AABAACAADAABAABA", pat[] = "AABA"*
Output: *Pattern found at index 0, Pattern found at index 9, Pattern found at index 12*

6

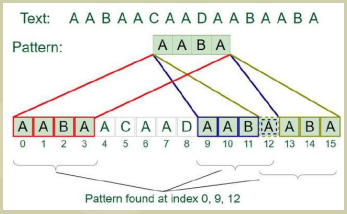
Técnicas de diseño de algoritmos

Manejo de strings

Naive

Algoritmo:

```
// C++ program for Naive Pattern Searching algorithm
#include <bits/stdc++.h>

using namespace std;

/* Función search(patrón, texto)
Devuelve el índice de dónde se encuentra, dentro del string, el inicio del patron */

void search(char* pat, char* txt) {
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++) {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j== M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            cout << "Pattern found at index " << i << endl;
    }
}
```



Output

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Time Complexity: $O(N^2)$
Auxiliary Space: O(1)

```
// Code
int main() {
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";

    // Function call
    search(pat, txt);
    return 0;
}
```

Naive algorithm for Pattern Searching. https://www.geeksforgeeks

7

Técnicas de diseño de algoritmos

Manejo de strings

Naive

Coincidencias de cadenas

- Cadena de texto T[0..N-1] — T = "abacaabaccabacabaabb"
- Patrón de la cadena P[0..M-1] — P = "abacab"
- ¿Dónde está la primera instancia de P en T? — T[10..15] = P[0..5]
- Normalmente, N >>> M

```
abacaabaccabacabaabb
abacab
 abacab
  abacab
   abacab
    abacab
     abacab
      abacab
       abacab
        abacab
         abacab
          abacab
```

- **Algoritmo de fuerza bruta**
  22 + 6 = 28 comparaciones.

8

Técnicas de diseño de algoritmos

Manejo de strings

Naive

Mejor caso

- Cuando el primer carácter del patrón no existe en el texto.

```
txt[] = "AABCCAADDEE";
pat[] = "FAA";
```

Complejidad: O(N)

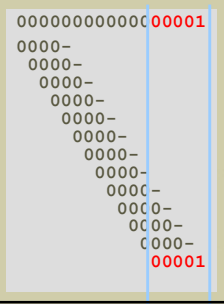
Peor caso

- Cuando todos los caracteres en el texto y en el patrón son iguales.

```
txt[] = "AAAAAAAAAAAAAAAAAA";
pat[] = "AAAAA";
```

- Cuando solamente el último carácter es diferente. Complejidad: O(M * (N – M + 1)).

```
txt[] = "AAAAAAAAAAAAAAAAAB";
pat[] = "AAAAB";
```



- 60+5 = **65** comparaciones
- *¿Cuántas de ellas se pudieran evitar?*

Naive algorithm for Pattern Searching, https://www.geeksforgeeks

Técnicas de diseño de algoritmos

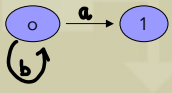
Manejo de strings

Máquina de estados finitos (FSM)

- La FSM es una máquina computacional que toma:
  - Una cadena como entrada
  - Salidas con respuesta SÍ/NO
    - Es decir, la máquina "acepta" o "rechaza" la cadena.

Modelo

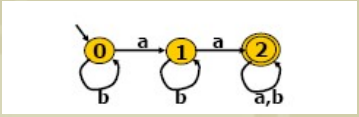
- Entrada a una FSM
  - Cadenas construidas a partir de un alfabeto fijo {a,b,c}
  - Posibles entradas: aa, aabbcc, a, etc.
- La máquina
  - Un grafo dirigido
    - Nodos = Estados de la máquina
    - Bordes = Transición de un estado a otro



- Estados especiales
  - Inicio (q_0) y Final (o Aceptación) (q_n)

Ejemplo:

- Supongamos que el alfabeto es {a,b}
  - ¿Qué cadenas acepta esta FSM?



Técnicas de diseño de algoritmos

Manejo de strings

Máquina de estados finitos (FSM)

- La FSM es una máquina computacional que toma:
  - Una cadena como entrada
  - Salidas con respuesta SÍ/NO
    - Es decir, la máquina “acepta” o “rechaza” la cadena.

Ejercicios:

Con el alfabeto {0, 1},

1) Dibuja un autómata finito que acepte cualquier cadena con un número "par" de unos.

2) Dibuja un autómata finito que acepte cualquier cadena con un número "par" de unos consecutivos seguido de un número "impar" de ceros consecutivos.

Técnicas de diseño de algoritmos

Manejo de strings

Máquina de estados finitos (FSM)

- La FSM es una máquina computacional que toma:
  - Una cadena como entrada
  - Salidas con respuesta SÍ/NO
    - Es decir, la máquina “acepta” o “rechaza” la cadena.

¿Por qué estudiar las máquinas de estados finitos?

- Técnica útil de diseño de algoritmos
  - Análisis léxico (“tokenización”)
  - Sistemas de control
    - Ascensores, Máquinas de refrescos…
- Modelar un problema con FSM es
  - Simple
  - Elegante

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n = T.length
2  q = 0
3  for i = 1 to n
4      q = δ(q, T[i])
5      if q == m
6          print "Pattern occurs with shift" i − m
```

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3      for each character a ∈ Σ
4          k = min(m + 1, q + 2)
5          repeat
6              k = k − 1
7          until P_k ⊐ P_q a
8          δ(q, a) = k
9  return δ
```

Técnicas de diseño de algoritmos

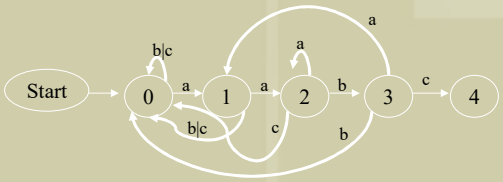
Manejo de strings

Máquina de estados finitos (FSM)

- La FSM es una máquina computacional que toma:
  - Una cadena como entrada
  - Salidas con respuesta SÍ/NO
    - Es decir, la máquina "acepta" o "rechaza" la cadena.

Usando el concepto FSM en la coincidencia de patrones

- Considere el alfabeto {a,b,c}
  - Supongamos que buscamos el patrón "aabc"
    - El autómata finito para "aabc" sería:




```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n = T.length
2  q = 0
3  for i = 1 to n
4      q = δ(q, T[i])
5      if q == m
6          print "Pattern occurs with shift" i − m
```


13

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

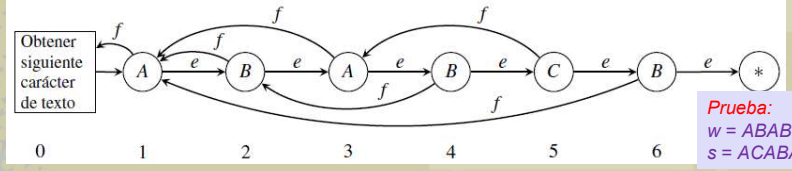
- Una mejor forma de resolver el problema de la coincidencia de un patrón es utilizar el algoritmo KPM, propuesto por Donal **Knuth**, James **Morris** y Vaughan **Pratt** en 1977.
- Algoritmo utilizado para determinar si una cadena *w* se encuentra contenida dentro de una cadena *s*, que puede ser de igual o mayor tamaño.

w = ABC
*s = ABAABB**ABC**ABD*

- Convierte la cadena de búsqueda en una máquina de estados finitos, luego ejecuta la máquina de estados con la cadena que se busca como dato de entrada.
- El tiempo de ejecución es O (m + n), donde *m* es la longitud de la cadena de búsqueda y *n* es la longitud de la cadena a buscar.

Ejemplo:

Diagrama de flujo KMP para *w*= '*ABABCB*'



Prueba:
w = ABABCB
s = ACABAABABA

14

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Idea básica

T= abcabx...
P= abcaby

- Se van comparando los caracteres a partir de la posición *i* del texto con los del patrón.
- Cuando uno coincide, en la subcadena del patrón que se tiene antes la NO coincidencia, buscamos el máximo prefijo que al mismo tiempo sea sufijo de la subcadena.
  - Si no existe, nos regresamos solo a la posición inicial de patrón.
  - Si existe un prefijo que es sufijo, y la longitud de este prefijo es *k*, continuamos a partir de la posición *i* + *k* del texto y *j* + *k* del patrón.
- Este proceso se realiza tan hacia atrás como sea necesario, pudiendo llegar a iniciar el patrón a su posición inicial.

P= abcaby

Ejemplo:

Si parte del texto es T = "abcabx...", y el patrón es P = "abcaby"

T= abcabx...
P=abcaby

LPS	0	1	2	3	4	5
	0	0	0	1	2	0

- Se inicia comparando los caracteres uno a uno, hasta que llegamos a la posición 6.
- En esta posición falla porque tenemos una ***x*** en el texto y una ***y*** en el patrón.
- En el método ingenuo tendríamos que iniciar todo desde la posición 2 del texto y la 1 del patrón.
- Pero, se observa que en la subcadena anterior a la falla, es decir, en "*abcab*", existe el prefijo *ab*, con longitud 2, que al mismo tiempo es sufijo de la misma cadena (*abcab*), lo que nos dice qué es seguro que los 2 caracteres anteriores al fallo **coinciden** completamente con el inicio del patrón, por lo que las siguientes comparaciones se pueden hacer a partir de la 6 del texto y la posición 3 del patrón.

15

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Pre-procesamiento

T= abcabx...
P=abcaby

LPS	0	1	2	3	4	5
	0	0	0	1	2	0

- Para poder hacer todo lo anterior en tiempo lineal, es necesario hacer un pre-procesamiento del patrón.
  - Este pre-procesamiento se hace usando un arreglo que inicia en la posición 1 (que sería la posición inicial de la cadena).
  - Al final de este proceso, el arreglo contiene la longitud del máximo prefijo que es también sufijo en la subcadena *P[1..i]*.

Algoritmo:

1) Defina una matriz unidimensional con un tamaño igual a la longitud del patrón. (LPS[tamaño])
2) Defina las variables *i* y *j*. Establezca i = 0, j = 1 y LPS[0] = 0.
3) Compare los caracteres en **Patrón[i]** y **Patrón[j]**.
4) Si ambos coinciden, establezca **LPS[j] = i+1** e incremente ambos valores de *i* y *j* en uno. Ve al paso 3.
5) Si no coinciden, verifique el valor de la variable 'i'.
   Si es '0', establezca **LPS[j] = 0** e incremente el valor de '*j*' en uno; si no es '0', establezca *i = LPS[i-1].*
   Ve al paso 3.
6) Repetir los pasos anteriores hasta que se completen todos los valores de LPS[].

Usamos la tabla LPS para decidir cuántos caracteres se omitirán para comparar cuando se produzca una discrepancia.

Algoritmo LPS:

```
m = |P|; // length P
Define an array LPS of size m
  LPS[0] = 0;
  i = 1; j = 0;
  while (i < m) {
    compare P[i] and P[j];
    if (P[j]==P[i])
      { LPS[i] = j+1; i++; j++; }
    else if (j>0) j=LPS[j-1];
    else {LPS[i] = 0; i++;}
  }
```

Usa valores previous de **LPS**

16

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Pre-procesamiento

Algoritmo:

1) Defina una matriz unidimensional con un tamaño igual a la longitud del patrón. (LPS[tamaño])
2) Defina las variables *i* y *j*. Establezca i = 0, j = 1 y LPS[0] = 0.
3) Compare los caracteres en **Patrón[i]** y **Patrón[j]**.
4) Si ambos coinciden, establezca **LPS[j] = i+1** e incremente ambos valores de *i* y *j* en uno. Ve al paso 3.
5) Si ambos no coinciden, verifique el valor de la variable '*i*'. Si es '0', establezca **LPS[j] = 0** e incremente el valor de '*j*' en uno; si no es '0', establezca *i = LPS[i-1]*. Ve al paso 3.
6) Repetir los pasos anteriores hasta que se completen todos los valores de LPS[].

T= abcabx...
P=abcaby

LPS	0	1	2	3	4	5
	0	0	0	1	2	0

Algoritmo LPS:

```
m = |P|; // length P
Define an array LPS of size m
 LPS[0] = 0;
 i = 1; j = 0;
 while (i < m) {
  compare P[i] and P[j];
  if (P[j]==P[i])
   { LPS[i] = j+1; i++; j++; }
  else if (j>0) j=LPS[j-1];
  else {LPS[i] = 0; i++;}
 }
```

Ejemplo:

- Considera el siguiente patrón de texto:

0	1	2	3	4	5	6
A	B	C	D	A	B	D

1) i = 0, j = 1 y LPS[0] = 0.
2) Compara el **Patrón[i]** y **Patrón[j]**.
   Si ambos coinciden, establezca **LPS[j] = i+1, i++, j++**. Ir paso 2.
3) Si no coinciden, entonces
   Si i == 0 entonces **LPS[j] = 0, j++**
   de lo contrario ***i = LPS[i-1]***. Ir al paso 2.
4) Repetir los pasos anteriores hasta que se completen todos los valores de LPS[].

LPS

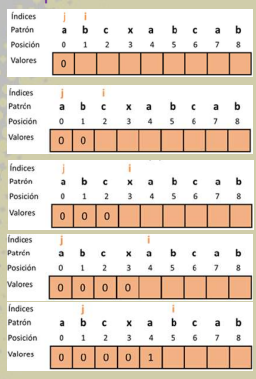
0	1	2	3	4	5	6
A	B	C	D	A	B	D

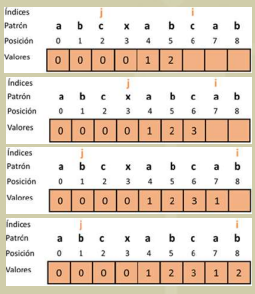
Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Pre-procesamiento





Algoritmo LPS:

```
m = |P|; // length P
Define an array LPS of size m
 LPS[0] = 0;
 i = 1; j = 0;
 while (i < m) {
  compare P[i] and P[j];
  if (P[j]==P[i])
   { LPS[i] = j+1; i++; j++; }
  else if (j>0) j=LPS[j-1];
  else {LPS[i] = 0; i++;}
 }
```

Una vez preprocesado el patrón, el algoritmo es sencillo dado que el arreglo LPS indica, *una vez que se encuentra un fallo*, a qué posición se debe regresar el patrón para seguir comparando con el texto.

Complejidad

- La complejidad del algoritmo resultante es lineal sobre la longitud del texto, *O(n)*.
- Y como la complejidad del pre-procesamiento es lineal sobre la longitud del patrón, *O(m)*.
- La complejidad final del algoritmo KMP es lineal sobre los procesos, *O(n + m)*, mucho más eficiente que el *O(nm)* del algoritmo ingenuo.

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Algoritmo KMP para buscar una palabra en un texto

```
Data: W: The word we are searching for
      T: The full text
      m: Length of W
      n: Length of T
      LPS: LPS array for word W
Result: Returns true if W is found inside T, or false otherwise
i ← 0;
j ← 0;
while i < n do
    while j ≥ 0 AND T[i] ≠ W[j] do
        j ← LPS[j];
    end
    i ← i + 1;
    j ← j + 1;
    if j = m then
        return true;
    end
end
return false;
```

output: if exists W in T return true/false

Algoritmo KMP

input: Text T and Pattern P

|T| = n

|P| = m

Compute Table LPS for Pattern P

i=j=0

```
while(i<n) {
  if(P[j]==T[i])
    { if (j == m-1) return i-m+1;
      i++; j++; }
  else if (j>0) j=LPS[j-1];
  else i++;
}
```

Use F to determine next value for j.

output: first occurrence of P in T

Ejemplo:

- Ubicar el patrón P = AABA, en el texto T = ABAABAACA.

T=	0	1	2	3	4	5	6	7	8
	A	B	A	A	B	A	A	C	A

P=	0	1	2	3
	A	A	B	A

LPS	0	1	2	3
	A	A	B	A

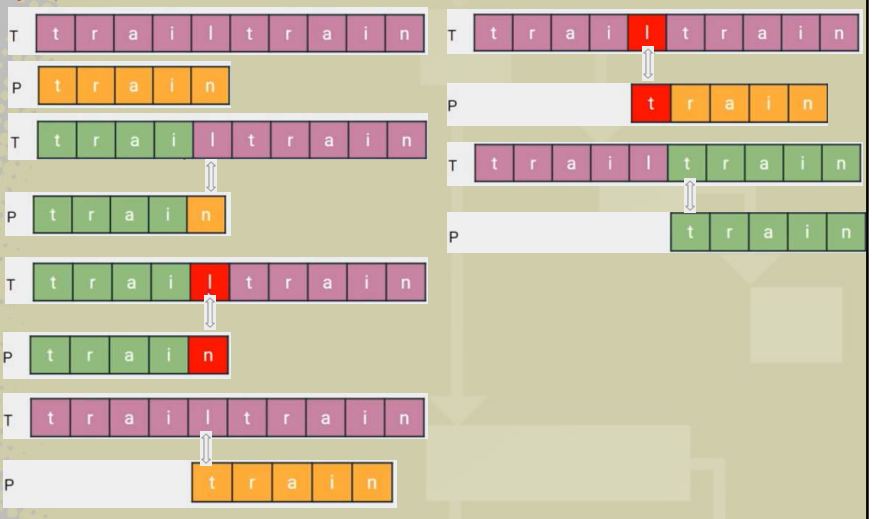
19

Técnicas de diseño de algoritmos

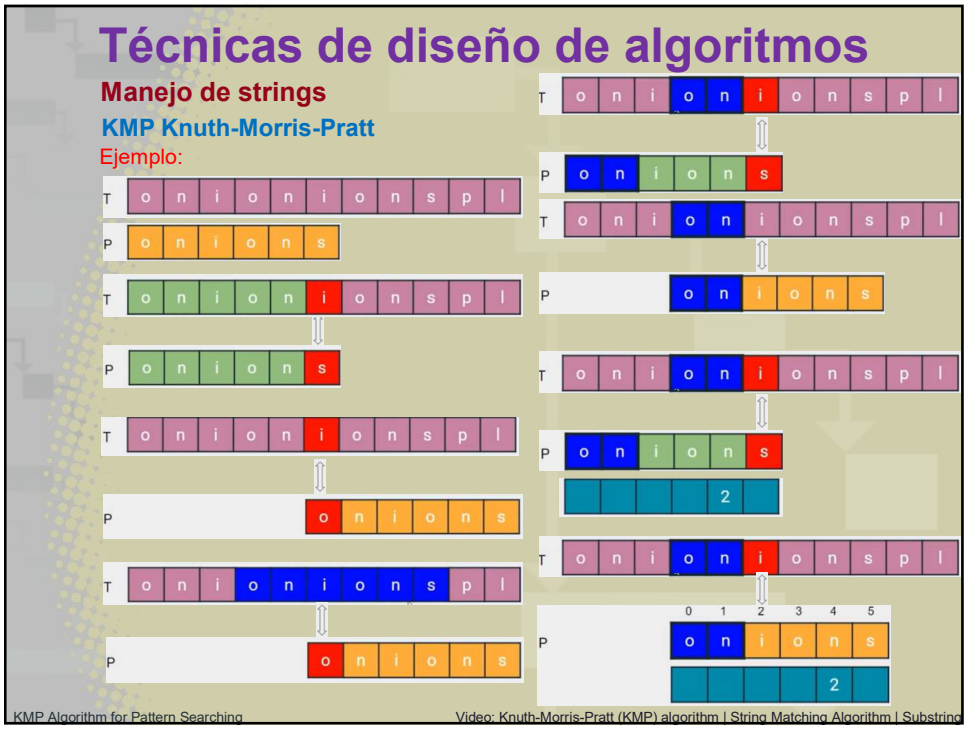
Manejo de strings

KMP Knuth-Morris-Pratt

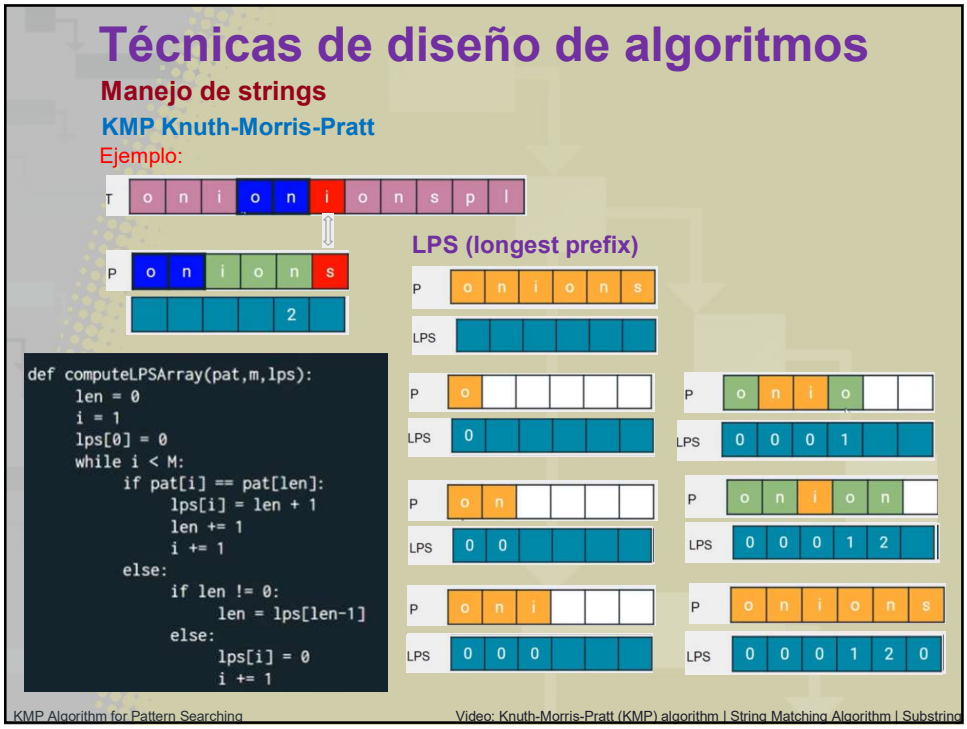
Ejemplo:



20


Técnicas de diseño de algoritmos
Manejo de strings
KMP Knuth-Morris-Pratt
Ejemplo:
KMP Algorithm for Pattern Searching
Video: Knuth-Morris-Pratt (KMP) algorithm | String Matching Algorithm | Substring

21


Técnicas de diseño de algoritmos
Manejo de strings
KMP Knuth-Morris-Pratt
Ejemplo:
LPS (longest prefix)
def computeLPSArray(pat,m,lps):
len = 0
i = 1
lps[0] = 0
while i < M:
if pat[i] == pat[len]:
lps[i] = len + 1
len += 1
i += 1
else:
if len != 0:
len = lps[len-1]
else:
lps[i] = 0
i += 1
KMP Algorithm for Pattern Searching
Video: Knuth-Morris-Pratt (KMP) algorithm | String Matching Algorithm | Substring

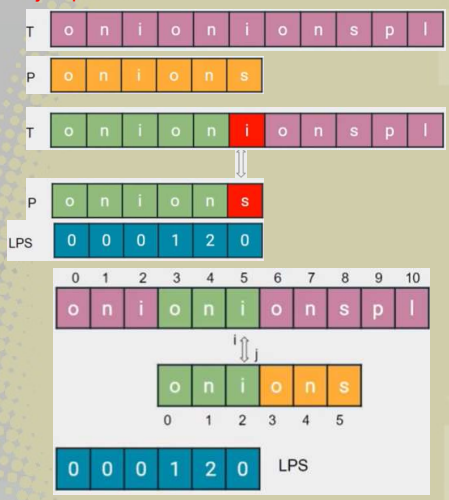
22

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Ejemplo:



Implementation - python

```
def KMPSearch(pat, txt):
    N = len(txt)
    M = len(pat)
    lps = [0]*M
    computeLPSArray(pat, M, lps)
    i=0
    j=0
    while i < N-M+1:
        if txt[i] == pat[j]:
            i += 1
            j += 1
        else:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
        if j == M:
            print(i-j)
            j = lps[j-1]
```

KMP Algorithm for Pattern Searching — Video: Knuth-Morris-Pratt (KMP) algorithm | String Matching Algorithm | Substring

23

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Ejemplo:

```
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
lps[] = {0, 1, 2, 3} i = 0, j = 0

i = 0, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 1, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 2, j = 2
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
pat[i] and pat[j] match, do i++, j++

i = 3, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 4, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3
Value of lps[j-1] gave us index of next character to match.

i = 4, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 5, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3

i = 5, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[2] = 2

i = 5, j = 2
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[1] = 1
Value of lps[j-1] gave us index of next character to mat

i = 5, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j > 0, change only j
j = lps[j-1] = lps[0] = 0
```

KMP Algorithm for Pattern Searching — Algorithms for Competitive Programming — EXAMPLE

24

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Ejemplo:

```
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
lps[] = {0, 1, 2, 3} i = 0, j = 0

i = 0, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 1, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 2, j = 2
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
pat[i] and pat[j] match, do i++, j++

i = 3, j = 3
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++, j++

i = 4, j = 4
Since j == M, print pattern found and reset j,
j = lps[j-1] = lps[3] = 3
```

Value of lps[j-1] gave us index of next character to match.

```
i = 5, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] do NOT match and j is 0, we do i++.

i = 6, j = 0
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++ and j++

i = 7, j = 1
txt[] = "AAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++ and j++
```

Y así continua el proceso…

25

Técnicas de diseño de algoritmos

Manejo de strings

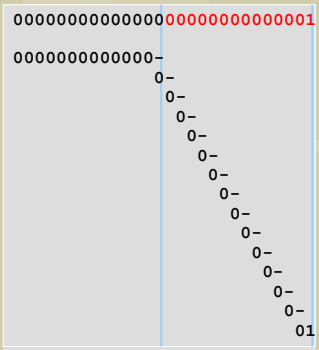
KMP Knuth-Morris-Pratt

Comparación

Fuerza bruta

- El peor caso:
  196 + 14 = 210 comparaciones

KMP



28+14 = 42 comparaciones

26

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Comparación

Fuerza bruta

```
abcdeabcdeabcedfghijkl

-
 bc-
  -
   -
    -
     -
      bc-
        -
         -
          -
           -
            bcedfg
```

21 comparaciones

KMP

```
abcdeabcdeabcedfghijkl

-
 bc-
    -
     -
      -
       bc-
          -
           -
            -
             bcedfg
```

19 comparaciones
5 comparaciones en preparación

Complejidad

- La complejidad del algoritmo resultante es lineal sobre la longitud del texto, *O(n)*.
- Y como la complejidad del pre-procesamiento es lineal sobre la longitud del patrón, *O(m)*.
- La complejidad final del algoritmo KMP es lineal sobre los procesos, *O(n + m)*, mucho más eficiente que el *O(nm)* del algoritmo ingenuo.

Técnicas de diseño de algoritmos

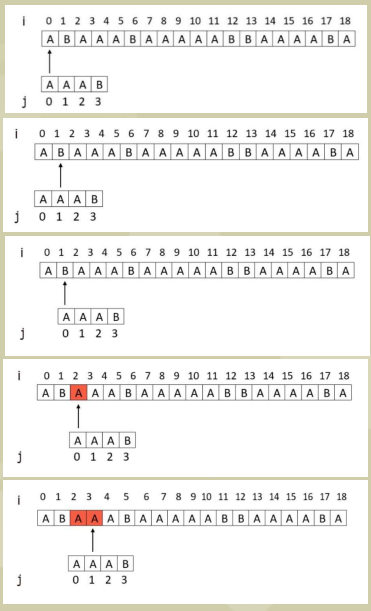
Manejo de strings

KMP Knuth-Morris-Pratt

Ejemplo:

S = "ABAAABAAAAABBAAAABA"
p = "AAAB"

1. i = 0, j = 0:
   S[i] and p[j] match so i++ and j++
2. i = 1, j = 1: **(Mismatch)**
   S[i] != p[j] and j != 0, so j = lps[j-1] → j = lps[0] = 0
3. i = 1, j = 0:
   S[i] != p[j] and j = 0, so i++
4. i = 2, j = 0:
   S[i] and p[j] **match** so i++ and j++
5. i = 3, j = 1:
   S[i] and p[j] **match** so i++ and j++



Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Ejemplo:

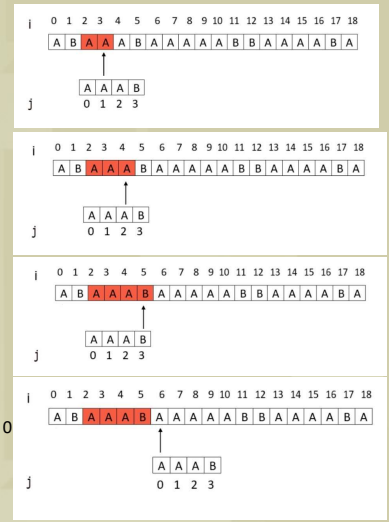
S = "ABAAABAAAAABBAAAABA"
p = "AAAB"

5. i = 3, j = 1:
   S[i] and p[j] **match** so i++ and j++

6. i = 4, j = 2:
   S[i] and p[j] **match** so i++ and j++

7. i = 5, j = 3:
   S[i] and p[j] **match** so i++ and j++

8. i = 6, j = 4:
   j = m, so **print location of pattern**
   es decir, (i-j) and j = lps[j-1]
   → Pattern found at location: (6-4) = 2, j = lps[3] = 0



29

Técnicas de diseño de algoritmos

Manejo de strings

KMP Knuth-Morris-Pratt

Ejercicios:

- Supongamos que tenemos el patrón P = 10010001 y el texto T = 000100100100010111
- Realiza lo siguiente:
  - Dibuja una máquina de estados para el patrón P.
  - Construye la tabla KMP para P.
  - Muestra el seguimiento del algoritmo KMP con T.

30