Semana 4

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

08-2021

Palíndromo más largo
- Definición
- Algoritmo de fuerza bruta
- Algoritmo de Manacher

Funciones Hash
- Polynomial Rolling Hash Function
- Arreglos de Sufijos (Suffix Array)

Palíndromo más largo

Un palíndromo es una cadena que se lee igual de izquierda a derecha que de derecha a izquierda. En algunas aplicaciones es necesario encontrar las subcadenas que son palíndromos, dentro de un string dado, o el palíndromo más grande que exista como subcadena de un cadena dada. El problema del palíndromo más largo y se define así:

- Dada una cadena S, se desea encontrar la subcadena de la misma que forma un palíndromo y que es el más largo que se pueda encontrar, es decir, el que está formado con el mayor número de caracteres.

Algoritmo de fuerza bruta

La forma más simple de resolver el problema del palíndromo más largo de una cadena S es usar fuerza bruta, en donde se recorren todas las posiciones i de la cadena, y para cada una de ellas se trata de expandir hacia la izquierda y a la derecha hasta encontrar la subcadena más grande, lo cual se debe hacer tanto para cadenas de longitud impar como para longitud par. Complejidad $O(n^2)$.

Procedure 1 LONGEST_PALINDROME

Input: S : String

```
maxLong ← 1
startAt ← 0
for i ← 1 to S.length do
    /* LONGITUD IMPAR */
    NEXT_SLIDE
    /* LONGITUD PAR */
    NEXT_SLIDE
end for
return PAIR(startAt, maxLong)
```

```
/* LONGITUD IMPAR */
j ← 1
while (i − j) ≥ 1 and (i + j) <= n and S[i − j] = S[i + j] do
  j ← j + 1
end while
if (2 ∗ j) − 1 > maxLong then
  maxLong ← (2 ∗ j) + 1
  startAt ← i − j + 1
end if
```

```
/* LONGITUD PAR */
j ← 1
while (i − j − 1) ≥ 1 and (i + j) <= n and S[i − j − 1] = S[i + j] do
  j ← j + 1
end while
if (2 ∗ j) > maxLong then
  maxLong ← 2 ∗ j
  startAt ← i − j
end if
```

Algoritmo de Manacher

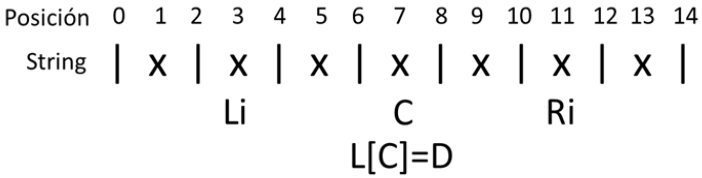
- Este algoritmo, propuesto por Glenn K. Manacher en 1975, logra resolver el problema del palíndromo más largo de una cadena S, en una forma muy eficiente, con una complejidad O(n), proporcional a la longitud n de la cadena.
- Al igual que el algoritmo a fuerza bruta, éste se basa en buscar un palíndromo centrado en i, con algunas diferencias:
  - En el algoritmo a fuerza bruta, se considera si la cadena es longitud par o impar. Por otra parte, el algoritmo de Manacher se basa en una nueva cadena, T, a la que se le agregan posiciones intermedias, una al inicio y otra al final de la cadena, lo que garantiza que todo el proceso se realizará sobre una cadena de longitud impar.
  - Los valores calculados se almacenan en un arreglo, L, que tiene la longitud de la nueva cadena.
  - Se utilizan los valores previamente calculados (a la izquierda de i) que están almacenados en L, para calcular los valores que siguen (a la derecha de i).

- El algoritmo inicial con la construcción de la nueva cadena, T, agregando un carácter especial en medio de cada uno de los caracteres de la cadena original, además de una al inicio y otro al final.
- El carácter que se agrega puede ser cualquiera que no pertenezca al alfabeto de la cadena analizada.
- Por ejemplo,
  - Si tenemos la cadena $S = baab$, de longitud 4, al agregar el carácter "|", la nueva cadena sería $T = |b|a|a|b|$, la cual tiene longitud de 9.
  - Si tenemos la cadena $S = bacab$, de longitud 5, al agregar el carácter "|", la nueva cadena sería $T = |b|a|c|a|b|$, la cual tiene longitud de 11.

Al finalizar el algoritmo, el valor en la posición i del arreglo L indica la longitud, en número de caracteres, que tiene el palíndromo más grande centrado en i, medido a la derecha o a la izquierda de i.

Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
String	|	c	|	a	|	c	|	b	|	c	|	a	|	c	|
L	0	1	0	3	0	1	0	7	0	1	0	3	0	1	0

Manacher se dio cuenta que la propiedad de simetría de un palíndromo podría ayudar en el cálculo de L aprovechando los valores que ya se habían calculado anteriormente. Sin embargo, las condiciones que se deben cumplir para poder disminuir la cantidad de cálculos no son triviales.



Notación:

- **C**: La última posición calculada de L. Todos los valores en posiciones menores o iguales en C en L se conocen.
- R_i: Posición a la derecha de C a la cuál se le quiere calcular su valor L.
- L_i: Posición a la izquierda de C que está separada exactamente el mismo número de caracteres de C que R_i. Esto implica que $R_i - C = C - L_i$.

Manacher encontró que sólo había 4 casos posibles y al implementarlos se reduce la complejidad. Es importante recordar que, dado que la longitud de la cadena original es n, la último posición de la cadena aumentada es $2n$.

1. Si $L[L_i] < (C + L[C]) - R_i$, entonces $L[R_i] = L[L_i]$.
2. Si $L[L_i] = (C + L[C]) - R_i$ y $(C + L[C]) = 2n$, entonces $L[R_i] = L[L_i]$.
3. Si $L[L_i] = (C + L[C]) - R_i$ y $(C + L[C]) < 2n$, entonces en este caso, no conocemos exactamente el valor de que debemos colocar en $L[R_i]$. Sin embargo, sabemos que ese valor debe ser al menos $L[L_i]$ y expandimos el palíndromo carácter por carácter.
4. Si $L[L_i] > (C + L[C]) - R_i$, entonces en este caso tampoco conocemos exactamente el valor de que debemos colocar en $L[R_i]$. Sin embargo, sabemos que ese valor debe ser al menos $L[L_i]$ y expandimos el palíndromo carácter por carácter.

Procedure 2 MANACHER

Input: S : String

$T \leftarrow EXPAND(S)$
$L \leftarrow Array[T.length], L[0] \leftarrow 0, L[1] \leftarrow 1$
$maxLong \leftarrow 0, maxCenter \leftarrow 0$
$C \leftarrow 1, R_i \leftarrow 0, L_i \leftarrow 0$
for $R_i \leftarrow 2$ **to** $T.length - 1$ **do**
  $L_i \leftarrow C - (R_i - C)$
  $expansion \leftarrow FALSE$
  /* is R_i within the bounds of the array? */
  if $(C + L[C]) - R_i \geq 0$ **then**
    /* SEE NEXT SLIDE */
  end if
end for

---

if $(C + L[C]) - R_i \geq 0$ **then**
  if $L[L_i] < (C + L[C]) - R_i$ **then**
    $L[R_i] \leftarrow L[L_i]$ /* CASE 1 */
  else if $L[L_i] = (C + L[C]) - R_i$ y $(C + L[C]) = 2n$ **then**
    $L[R_i] \leftarrow L[L_i]$ /* CASE 2 */
  else if $L[L_i] = (C + L[C]) - R_i$ y $(C + L[C]) < 2n$ **then**
    $L[R_i] \leftarrow L[L_i], expansion \leftarrow TRUE$ /* CASE 3 */
  else if $L[L_i] > (C + L[C]) - R_i$ **then**
    $L[R_i] \leftarrow (C + L[C]) + R_i, expansion \leftarrow TRUE$ /* CASE 4 */
  end if
else
  $L[R_i] \leftarrow 0, expansion \leftarrow TRUE$
end if

---

if $expansion = TRUE$ **then**
  while $(R_i + L[R_i]) < T.length$ **and** $(R_i - L[R_i]) > 0$ **and** $T[R_i + L[R_i] + 1] = T[R_i - L[R_i] - 1]$ **do**
    $L[R_i] \leftarrow L[R_i] + 1$
  end while
end if
if $(R_i + L[R_i]) > (C + L[C])$ **then**
  $C \leftarrow R_i$
end if
if $L[R_i] > maxLong$ **then**
  $maxLong \leftarrow L[R_i], maxCenter \leftarrow R_i$
end if
return $PAIR((maxCenter - maxLong)/2, maxLong)$

Introducción

- Los algoritmos de hashing son muy útiles en muchas áreas de las ciencias computacionales, sobre todo porque, usando una buena función de hash, es posible reducir la búsqueda a un tiempo casi constante $O(1)$.
- Por ejemplo, dadas las cadenas S_1, de longitud n_1, y S_2, de longitud n_2, indicar si son iguales o no. Si realizamos la comparación a fuerza bruta, el algoritmo resultante tiene un orden $O(min(n_1, n_2))$. Sin embargo, usando una buena función de hash podríamos logar un orden $O(1)$.

- La idea principal de una función hash para cadenas es convertir una cadena en un número entero. Para esto, la función debe cumplir la condición de que, si dos cadenas son iguales, les debe asignar el mismo número entero. Cuando la función le asigna el mismo número a dos cadenas diferentes se dicen que hubo una **colisión**. La función ideal sería aquella que a cada cadena le asignara un entero diferente.
- En realidad, es muy complicado hacer una función con tal condición, pero es suficiente con generar una función donde la probabilidad de asignarle el mismo entero a dos cadenas diferentes sea muy baja, y ese tipo de funciones sí se han podido generar.
- Un ejemplo es la llamada **Polynomial Rolling Hash Function**.

Polynomial Rolling Hash Function

Sea S, una cadena de longitud n, a la que se le quiere aplicar la función hash, y sean $S[0]$, $S[1]$, ..., $S[n-1]$ las representaciones en enteros de cada uno de los caracteres que lo forman, la función Polynomial Rolling Hash Function para la cadena S se define como:

$$\begin{aligned} prhs(S) &= (S[0] * p^0 + S[1] * p^1 + ... + S[n-1] * p^{n-1}) \mod m \\ &= (\sum_{i=0}^{n-1} S[i] * p^i) \mod n \end{aligned}$$

Donde, p y m son número enteros positivos que se deben seleccionar. Una forma común es que ambos sean números enteros primos.

- En el caso de P, se recomienda que sea un número primo cercano al número de caracteres en el alfabeto. Por ejemplo, para los caracteres en español, que son 27, el número primo más cercano es el 31, por lo que $p = 31$ es una buena opción. Si se incluyen también las mayúsculas, el conjunto sube a 54 y entonces $p = 53$ es una buena opción.
- En lo que respecta a m, debe ser un número muy grande, debido a que la probabilidad de que haya colisión es, aproximadamente, 1/m. Una buena selección ha sido un número primo muy grande, por ejemplo, $m = 10^9 + 9$, lo que nos da una probabilidad de colisión de alrededor de 10^{-9}.

- Si bien, una probabilidad de colisión de 10^{-9} es muy baja, si se hace una sola comparación. En algunos problemas es necesario comparar una cadena con un conjunto de otras cadenas. Si ese conjunto es de 10^6 elementos, la probabilidad de que suceda una colisión aumenta a 10^{-3}, y si se comparan las 10^6 cadenas entre si, la probabilidad se acerca peligrosamente a 1.
- Una forma muy común y simple de obtener mejores probabilidades de colisión, incluso con muchas comparaciones, es hacer dos funciones hash, con diferentes p y m. Se aplican las dos funciones hash y sólo se declaran iguales las cadenas si el resultado de ambas funciones son iguales. En este caso, si m para la segunda función es también cercana a 10^9, aplicar las dos funciones es equivalente a tener una m aproximadamente igual a 10^{18}.

```
long long prhf(const std::string &s) {
  int p = 31;
  int m = 1e9 + 9;
  long long result = 0;
  long long power = 1;

  for (int i = 0; i < s.size(); i++) {
    result = (result + ((s[i] - 'a' + 1) * power)) % m;
        power = (power * p) % m;
  }
  return power
}
```

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.

Procedure 3 SUFFIX_ARRAY

FORMAR LA CADENA DE DE TRABAJO $T \leftarrow S\$$
FORMAR TODOS LOS SUFIJOS DE T
ORDENAR LEXICOGRÁFICAMENTE LOS SUFIJOS ENCONTRADOS CONSIDERANDO A $ COMO PRIMER SÍMBOLO DEL ALFABETO
FORMAR EL ARREGLO A CON LAS POSICIONES DE INICIO EN LA CADENAS T DE CADA UNO DE LOS SUFIJOS
return A