Tecnológico de Monterrey

Evidencia 1: Actividad Integradora

Jessica Odette Guizado Hernández A01643724

Luz Patricia Hernández Ramírez A01637277

Análisis y diseño de algoritmos avanzados

Grupo 602

domingo 09 de septiembre, 2024

Evidencia 1: Actividad Integradora

Algoritmo KMP(Knuth-Morris-Pratt) $O(n + m)$

Se basa en recordar hasta qué punto han coincidido los caracteres entre el patrón y el texto, y cuando hay una discordancia, vuelve a la última posición donde hubo coincidencia y aprovecha esa información para saltar partes del patrón que ya no es necesario comparar de nuevo. Esto lo hace utilizando el array LPS (longest prefix suffix).

- Prefijo: Subcadena que va desde el inicio del patrón hasta antes del final.
- Sufijo: Subcadena que empieza después del primer carácter y termina en el final del patrón.

¿Cómo funciona el LPS? Almacena la longitud del prefijo más largo que también es un sufijo de una subcadena del patrón.
Ejemplo:

```
patrón = "ABABC"

LPS = [0, 0, 1, 2, 0]

A -> No hay prefijo ni sufijo posibles porque solo hay un carácter.LPS[0]=0

AB ->
Prefijos: "" (cadena vacía) y "A".
Sufijos: "B" y "" (cadena vacía).
Ningún prefijo coincide con un sufijo. LPS[1]=0

ABA ->
Prefijos: "", "A" y "AB".
Sufijos: "A", "BA" y "ABA".
El prefijo "A" coincide con el sufijo "A". LPS[2] = 1.

ABAB ->
Prefijos: "", "A", "AB" y "ABA".
Sufijos: "B", "AB", "BAB" y "ABAB".
El prefijo "AB" coincide con el sufijo "AB". LPS[3] = 2.

ABABC ->
Prefijos: "", "A", "AB", "ABA" y "ABAB".
Sufijos: "C", "BC", "ABC", "BABC" y "ABABC".
Ningún prefijo coincide con un sufijo.LPS[4]=0
```

Cuando hay una discordancia, en lugar de retroceder en el texto, el algoritmo "salta" a una posición del patrón que sigue siendo compatible con las coincidencias anteriores, utilizando la información del LPS para continuar la búsqueda eficientemente.

Algoritmo Manacher $O(n)$

Es un algoritmo eficiente para encontrar el palíndromo más largo dentro de una cadena, su objetivo principal es encontrar la subcadena más larga dentro de una cadena dado que sea un palíndromo.
¿Cómo funciona?

- Se inserta un carácter especial, como #, entre cada carácter del string para tratar los palíndromos de longitud par e impar de la misma manera.
- Esto asegura que todos los palíndromos sean de longitud impar.
- Se realiza un proceso de propagación del radio del palíndromo:

P[]: Almacena el radio del palíndromo más largo en cada posición.

```
Si tienes una cadena original "abac", se preprocesa como "#a#b#a#c#".
El array P[] queda:
P = [0, 1, 0, 3, 0, 1, 0, 0, 0]
Esto significa:
En la posición 1 (dónde está a), hay un palíndromo de radio 1: "a".
En la posición 3 (dónde está b), hay un palíndromo de radio 3: "aba".
En otras posiciones, el radio es menor o no hay palíndromos más largos.
```

- Se establecen las variables center y right: Mantienen el centro y el borde derecho del palíndromo más largo encontrado hasta el momento.

El algoritmo trata de expandir los palíndromos, si estamos dentro del palíndromo actual (es decir, `i < right`), podemos usar la simetría de los palíndromos:
El palíndromo centrado en la posición i tiene un "espejo" en la posición mirror, donde

```
mirror = 2 * center - i.
```

Si el radio en la posición mirror es más pequeño que la distancia entre i y right, entonces sabemos que el palíndromo en i será como mínimo tan largo como el palíndromo en mirror.
Si el carácter a la izquierda de i - P[i] es igual al carácter a la derecha de i + P[i], extendemos el palíndromo incrementando P[i]. Finalmente actualización de center y right.
El resultado se obtiene una vez que se ha calculado el array P[], el palíndromo más largo tiene el mayor valor en P[], y las posiciones del palíndromo más largo en la cadena original se calculan a partir de esa posición.
Ejemplo:

```
cadena = "abac"
cadena_preprocesada = "#a#b#a#c#"
P = [0, 1, 0, 3, 0, 1, 0, 0, 0]
center = 3
right = 6
El palíndromo más largo es "aba".
Las posiciones en la cadena original son 1 y 3 (donde la primera "a" está en la
posición 1 y la última "a" está en la posición 3).
```

Algoritmo LCS (O n*m)

El algoritmo de subsecuencia común más larga busca encontrar la subsecuencia más larga de dos secuencias que existen en los strings proporcionados y no necesita ser contigua en ambas cadenas y pueden formarse con letras o caracteres que no necesariamente son adyacentes. La forma en la que funciona el algoritmo es la siguiente:

1. Construimos una tabla vacía de adyacencia, esta tabla debe de n x m en donde n es la secuencia de X y m la secuencia de Y. Las filas en la tabla representan los elementos de la secuencia X y las columnas representan los elementos de la secuencia Y.

2. Las filas y columnas en la posición cero deben llenarse con ceros. Los valores restantes se llenan con base en diferentes casos. Y se comienza recorriendo la matriz .

    1. **Caso donde coinciden**: Si los caracteres transmission1[i-1] y transmission2[j-1] son iguales, se actualiza L[i][j] con el valor de L[i-1][j-1] + 1. Esto indica que el substring común más largo se ha extendido. Si esta nueva longitud es mayor que la max_len anterior, se actualizan max_len y end_pos con la longitud del substring y su posición final en transmission1.
    2. **Caso donde no coinciden**: Si los caracteres no coinciden, L[i][j] se establece en 0, ya que un substring común debe ser contiguo, y aquí se rompe la continuidad.

3. Una vez que la matriz esté llena, podemos reconstruir la LCS trazando el recorrido desde la esquina inferior derecha hacia la superior izquierda. Si el valor actual coincide con el de la celda en la diagonal superior izquierda, significa que se identificó un carácter común y se añade a la LCS.