Instituto Tecnológico de Estudios Superiores de Monterrey
Campus Ciudad de México

TC2038.651 Diseño y Análisis de Algoritmos Avanzados

Actividad Integradora 1

Alumno
Emiliano Saucedo Arriola | A01659258
A01659258@tec.mx

Profesor
Sergio Ruiz Loza

Fecha de entrega

08 de septiembre de 2022

Introducción

Hoy en día, el ser humano ha desarrollado e implementado una gran variedad de métodos de programación que nos permiten resolver problemas; sin embargo, no siempre resulta eficiente utilizar la misma técnica para analizar diferentes problemáticas. A pesar de que cada día contamos con equipos de cómputo más potentes, hay problemas que requieren de una gran cantidad de recursos computacionales, por lo que uno como desarrollador debe ser capaz de optimizar algoritmos robustos para brindar soluciones que sean eficientes mediante programas que posean una complejidad de bajo costo. Por lo anterior, el análisis y diseño de algoritmos, resulta ser una actividad compleja, ya que es necesario comprender el problema, así como determinar los pasos a seguir para que, de esta manera seamos capaces de llegar a una solución que sea eficiente y de calidad.

En la actividad presente, se busca dar solución a la situación problema, en la que se deben analizar archivos de texto de gran tamaño, para analizar la similitud de los archivos y encontrar el substring más largo común entre ambos archivos de transmisión.

Descripción de la evidencia

En equipos de máximo 3 personas, escribe un programa en C++ que lea 5 archivos de texto (de nombre fijo, no se piden al usuario) que contienen exclusivamente caracteres del 0 al 9, caracteres entre A y F y saltos de línea.

- mcode1.txt
- mcode2.txt
- mcode3.txt
- transmission1.txt
- transmission2.txt

Los archivos de transmision contienen caracteres de texto que representan el envío de datos de un dispositivo a otro. Los archivos mcodex.txt representan código malicioso que se puede encontrar dentro de una transmisión.

El programa debe analizar si el contenido de los archivos mcode1.txt, mcode2.txt y mcode3.,txt están contenidos en los archivos transmission1.txt y transmission2.txt y desplegar un true o false si es que las secuencias de chars están contenidas o no. En caso de ser true, muestra true, seguido de exactamente un espacio, seguido de la posición en el archivo de transmisiónX.txt donde inicia el código de mcodeY.txt.

Suponiendo que el código malicioso tiene siempre código "espejeado" (palíndromos de chars), sería buena idea buscar este tipo de código en una transmisión. El programa después debe buscar si hay código "espejeado" dentro de los archivos de transmisión (palíndromo a nivel chars, no meterse a nivel bits). El programa muestra en una sola linea dos enteros separados por un espacio correspondientes a la posición (iniciando en 1) en donde inicia y termina el código "espejeado" más largo (palíndromo) para cada archivo de transmisión. Puede asumirse que siempre se encontrará este tipo de código.
Finalmente el programa analiza que tan similares son los archivos de transmisión, y debe mostrar la posición inicial y la posición final (iniciando en 1) del primer archivo en donde se encuentra el substring más largo común entre ambos archivos de transmisión.

Solución

Como se puede observar, la Situación Problema se completa con 3 escenarios:

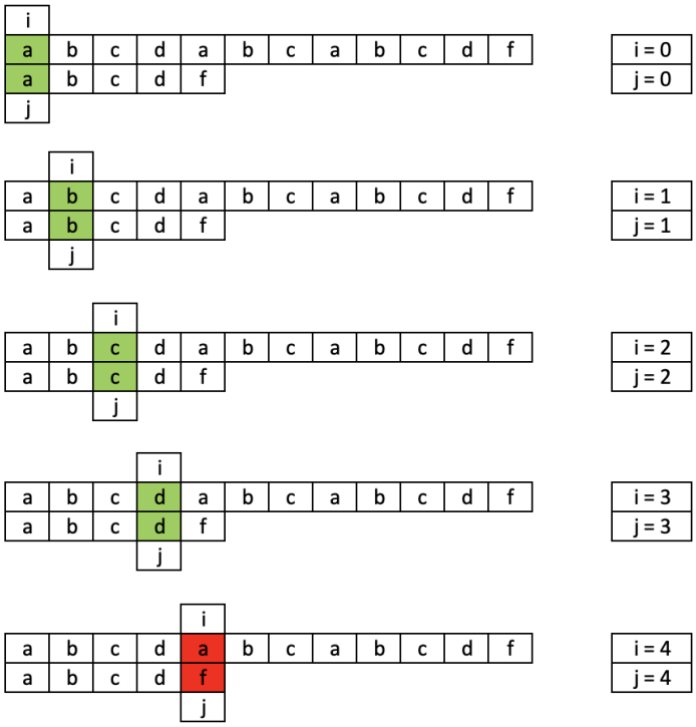
1) Búsqueda de substrings en una cadena

Para este primer punto, existen diversas formas de resolver el problema. De manera inmediata, podemos pensar en utilizar un algoritmo "ingenuo". Por ejemplo, consideremos los siguientes datos para visualizar como funcionaría este enfoque:

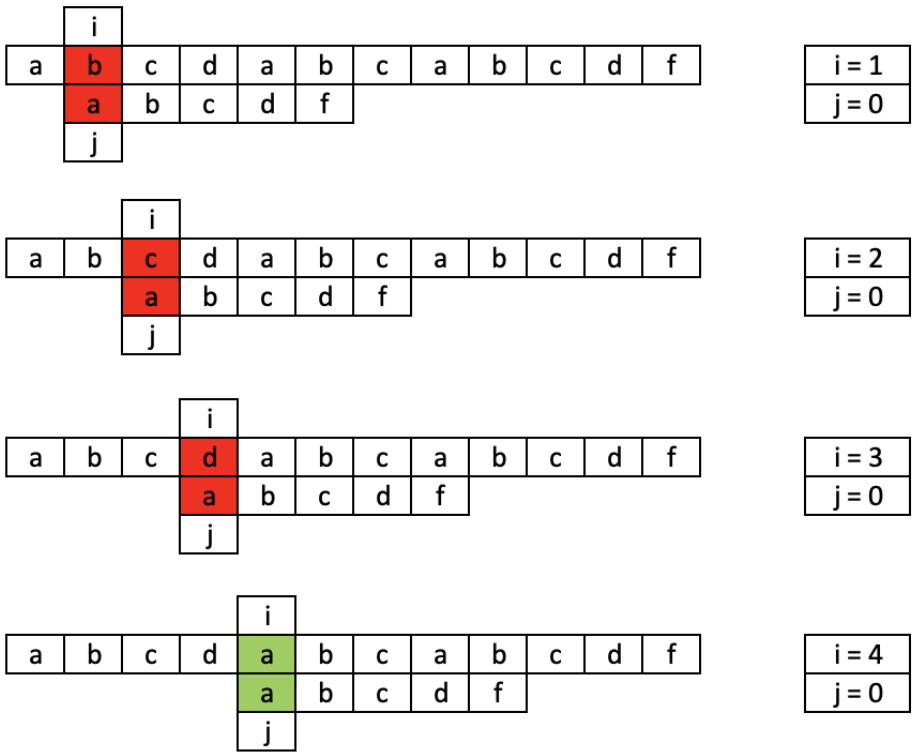
cadena	"abcdabcabcdf"
substring	"abcdf"

Este método contempla 2 iteradores (*i*, *j*). Uno de ellos se encargará de recorrer la cadena (*i*) y el otro el substring a buscar (*j*).

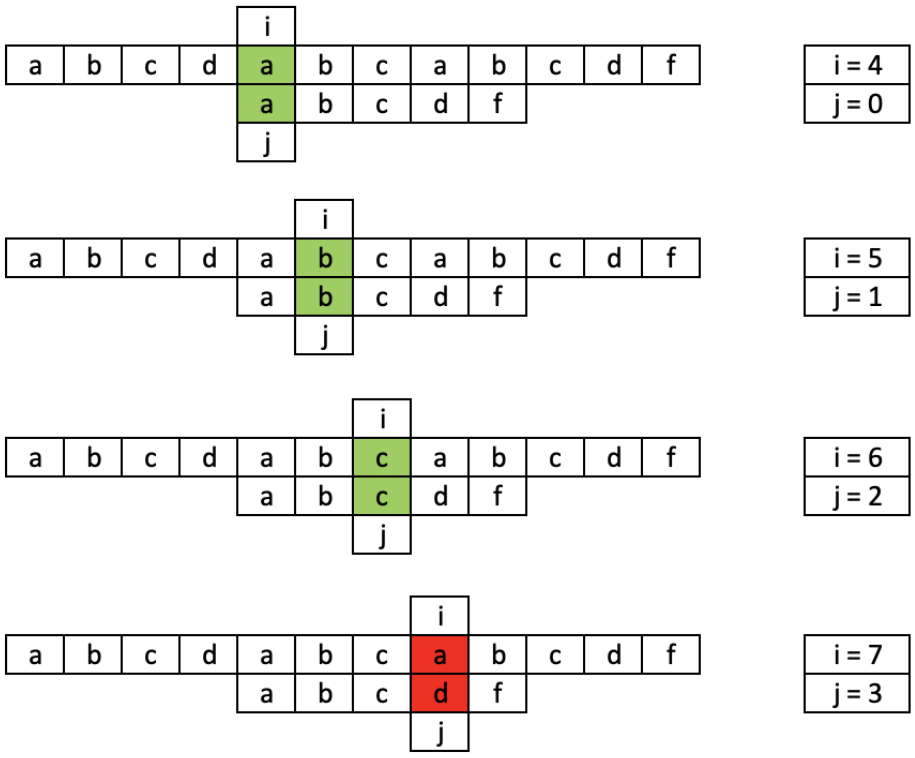
Ambos comienzan en el índice 0. Si el caractér en el que están posicionados ambos iteradores, en su respectivo string, es igual (cadena[*i*] == substring[*j*]), movemos ambos iteradores a la derecha (una posición). Si no son iguales, el iterador *j* empieza nuevamente desde el principio, y se compara con la cadena en el índice *i*, una posición a la derecha. En caso de que *j* logre recorrer el substring completo, significa que hemos encontrado el substring en la cadena:



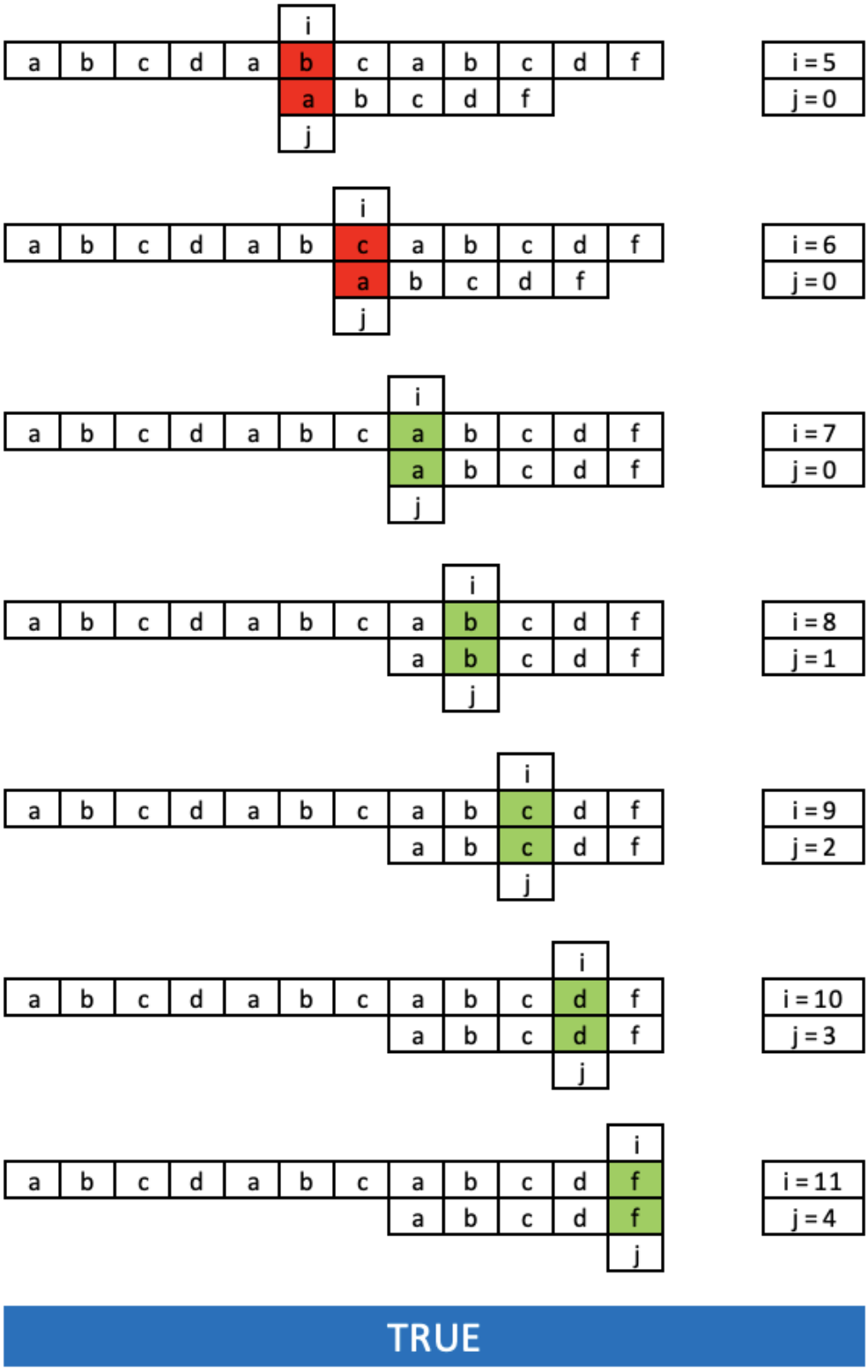
Como los índices no coinciden, *j* se reseteará en 0 y el iterador *i* se moverá una posición a la derecha con respecto a la comparación inicial, es decir, *i* = 0 + 1 = 1. Continuamos con el proceso



Encontramos nuevamente una coincidencia, por ello, *j* puede iterar en el substring simultáneamente con *i*.



Repetimos los pasos anteriores hasta encontrar el substring en la cadena, en caso de que exista, o bien, hasta que *i* ya no pueda iterar.



Para implementar esta solución en código, necesitaríamos de 2 ciclos for anidados de longitud *m* y *n*, lo que se traduce en una complejidad O(n*m). Dicha complejidad es muy cara, por lo que resulta conveniente una solución más eficiente. La solución propuesta es utilizar el algoritmo de búsqueda de substrings *KMP Search.*

El algoritmo Knuth-Morris-Pratt (KMP) es un algoritmo avanzado que aprovecha conceptos, como prefijos y sufijos, para hacer la búsqueda más eficiente.

Un *prefijo* es una subcadena de una cadena. El sentido es de izquierda a derecha.
Ejemplo:

- Cadena = abcde
- Prefijos = a, ab, abc, abcd

Un *sufijo* es una subcadena de una cadena. El sentido es de derecha a izquierda.
Ejemplo:

- Cadena = abcde
- Prefijos = e, de, cde, bcde

KMP compara los prefijos y sufijos de una cadena con el fin de encontrar "pequeñas" coincidencias para determinar donde se podría encontrar la siguiente existencia, sin la necesidad de analizar múltiples veces los caracteres de la cadena (en realidad, solo los analiza 1 vez). Ejemplo:

- Cadena = abcdabc
- Prefijos = a, ab, abc, abcd, …
- Sufijos = c, bc, abc, dabc, …

Para la implementación de esta solución, necesitamos considerar un preproceso: la tabla del prefijo sufijo más largo, también conocido como el Longest Prefix Suffix (LPS). Esta tabla representará lo previamente explicado.

Consideraciones:

- El índice 0 del vector lps siempre es 0.
- *i* = Posición actual. Inicia en 1.
- *j* = Posición previa. Inica en 0.

Funcionamiento:

- Necesitamos 2 iteradores: *i* & *j*.
- Si dos posiciones consecutivas coinciden, significa que sus sufijos y prefijos también lo hacen, por ello, lps en la posición actual es igual a lps en la posición previa + 1. Movemos ambos iteradores a la derecha.
- Si no coinciden recorremos el iterador *j* a la izquierda, con respecto el valor de lps[*j*-1], para buscar alguna coincidencia. No necesariamente disminuye 1 unidad.
- Si *j* == 0, se traduce en que no hubo coincidencias, por lo que lps en la posición actual es 0. Actualizamos el valor de *i*+=1 (a la derecha).

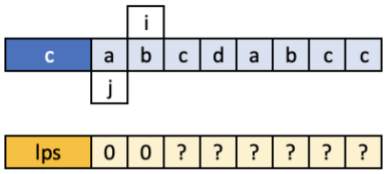
Ejemplo:

cadena	"abcdabcc"

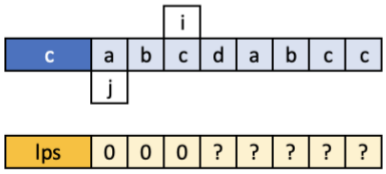
El vector lps es de la misma longitud que la cadena:

lps	0	?	?	?	?	?	?	?

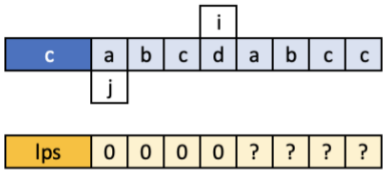
Considerando el funcionamiento previamente descrito, podemos determinar:



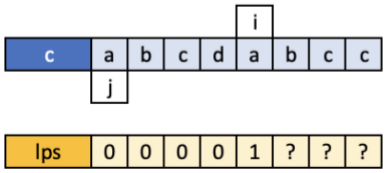
iteradores	i = 1	j = 0
caso	c[i] != c[j] && j == 0	
actualizar	i = 2	lps[i] = 0



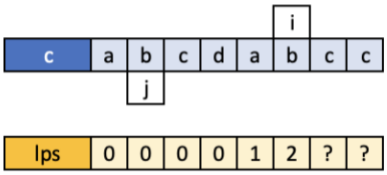
iteradores	i = 2	j = 0
caso	c[i] != c[j] && j == 0	
actualizar	i = 3	lps[i] = 0



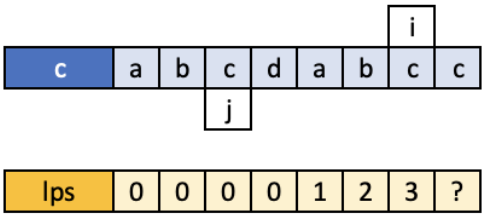
iteradores	i = 3	j = 0
caso	c[i] != c[j] && j == 0	
actualizar	i = 4	lps[i] = 0



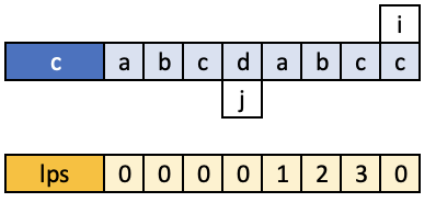
iteradores	i = 4	j = 0	
caso	c[i] == c[j]		
actualizar	i = 5	j = 1	lps[i] = lps[i-1] + 1



iteradores	i = 5	j = 1	
caso	c[i] == c[j]		
actualizar	i = 6	j = 2	lps[i] = lps[i-1] + 1

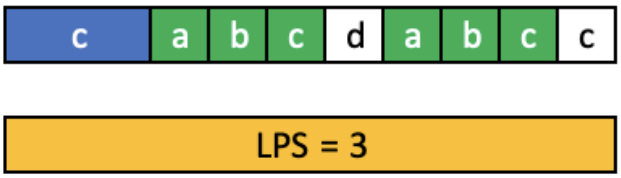


iteradores	i = 6	j = 2	
caso	c[i] == c[j]		
actualizar	i = 7	3	lps[i] = lps[i-1] + 1



iteradores	i = 6	j = 3	
caso	c[i] != c[j] && j != 0		
actualizar	i = 7	j = lps[3-1] = lps[2] = 0	lps[i] = 0

La importancia de dicha tabla es encontrar el valor más grande, ya que el mismo representa la coincidencia más grande. El resultado es el siguiente:



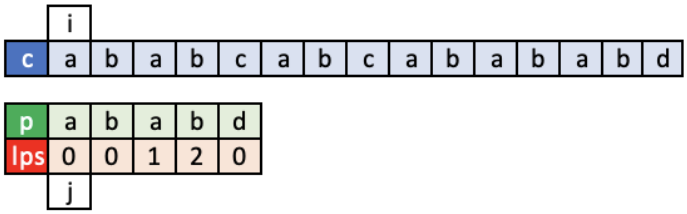
La complejidad de este preproceso es de O(n), debido a que, a pesar de que tenemos 2 iteradores, solo 1 de ellos tiene la función de analizar el string completo (*i*); la implementación en código requiere de 1 ciclo while.

El algoritmo KMP aplica este subproceso en el substring a buscar, por ello su eficiencia. Al tener acceso al arreglo que contiene las coincidencias, lo que sucede es que si hay un mismatch entre el substring y la cadena, no necesita resetear (de manera inmediata) ambos iteradores en 0; por el contrario, con apoyo de los prefijos y sufijos puede resolver el problema de manera más eficiente.

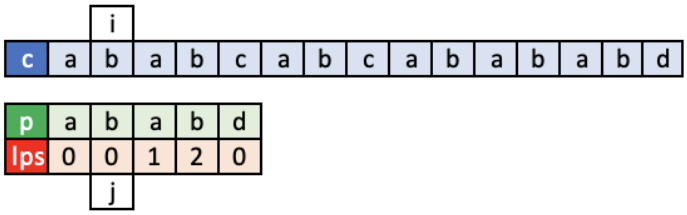
Considerando la siguiente información, aplicaremos el algoritmo KMP para encontrar el patrón en la cadena:

cadena	"ababcabcabababd"

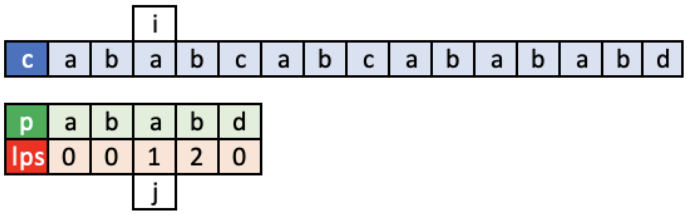
patrón	"ababd"
LPS del patrón	[0, 0, 1, 2, 0]



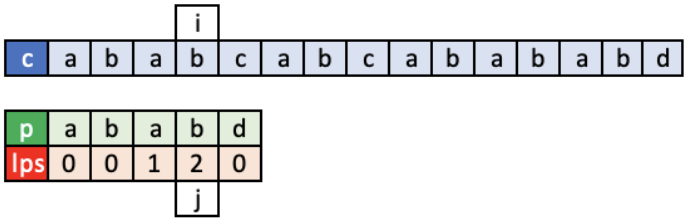
iteradores	i = 0	j = 0
caso	c[i] == p[j]	
actualizar	i = 1	j = 1



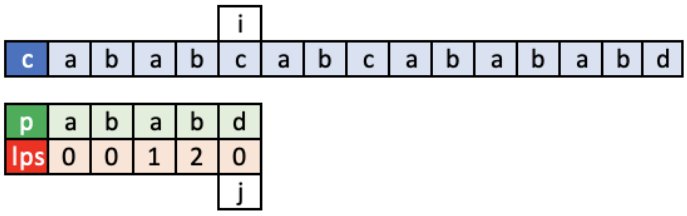
iteradores	i = 1	j = 1
caso	c[i] == p[j]	
actualizar	i = 2	j = 2



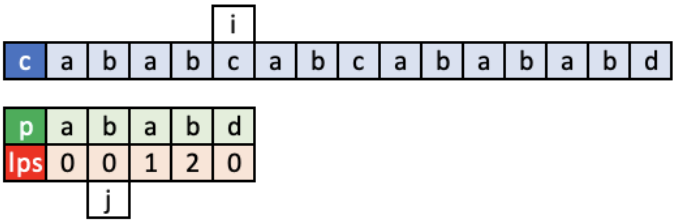
iteradores	i = 2	j = 2
caso	c[i] == p[j]	
actualizar	i = 3	j = 3



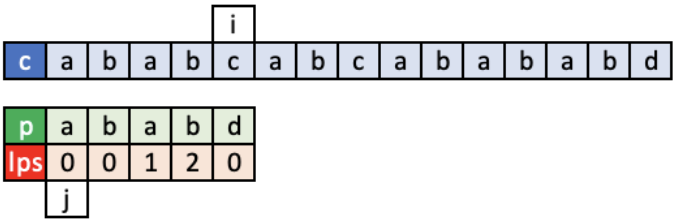
iteradores	i = 3	j = 3
caso	c[i] == p[j]	
actualizar	i = 4	j = 4



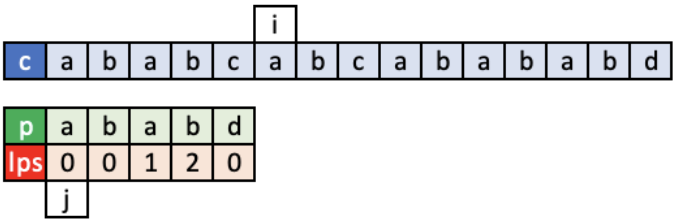
iteradores	i = 4	j = 4
caso	c[i] != p[j] && j != 0	
actualizar	j = lps[j-1] = lps[2] = 1	



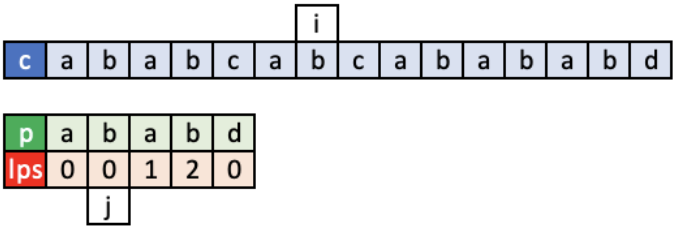
iteradores	i = 4	j = 1
caso	c[i] != p[j] && j != 0	
actualizar	j = lps[j-1] = lps[0] = 0	



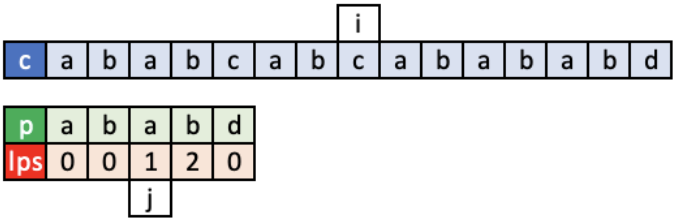
iteradores	i = 4	j = 0
caso	c[i] != p[j] && j == 0	
actualizar	i = 5	



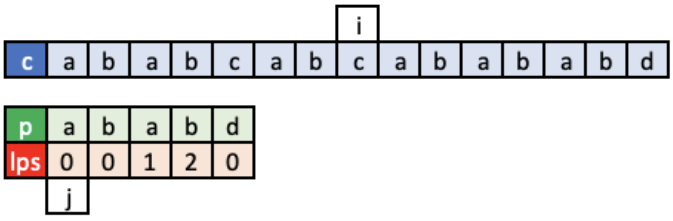
iteradores	i = 5	j = 0
caso	c[i] == p[j]	
actualizar	i = 6	j = 1



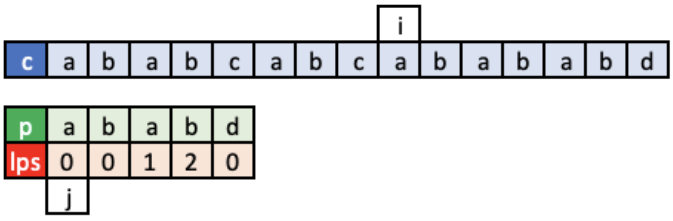
iteradores	i = 6	j = 1
caso	c[i] == p[j]	
actualizar	i = 7	j = 2



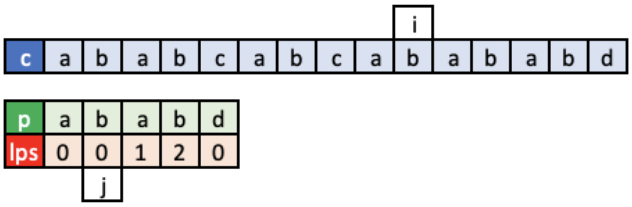
iteradores	i = 7	j = 2
caso	c[i] != p[j] && j != 0	
actualizar	j = lps[j-1] = lps[1] = 0	



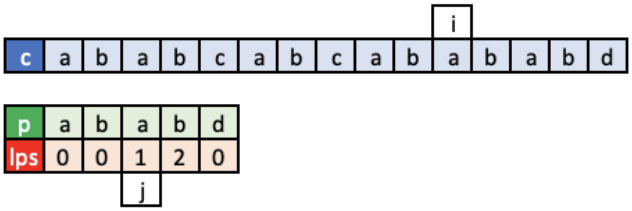
iteradores	i = 7	j = 0
caso	c[i] != p[j] && j == 0	
actualizar	i = 8	



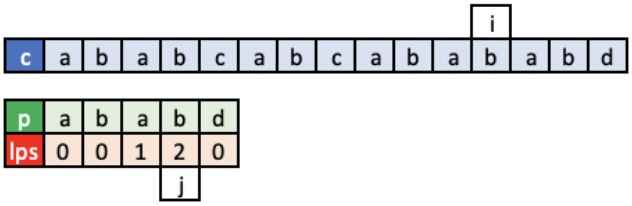
iteradores	i = 8	j = 0
caso	c[i] == p[j]	
actualizar	i = 9	j = 1



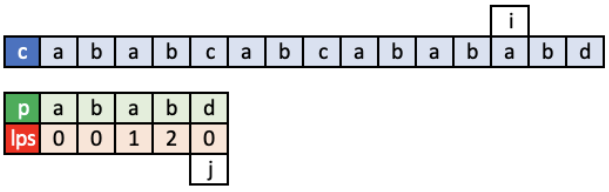
iteradores	i = 9	j = 1
caso	c[i] == p[j]	
actualizar	i = 10	j = 2



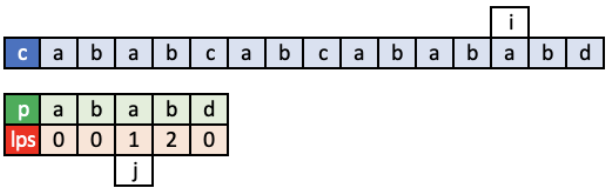
iteradores	i = 10	j = 2
caso	c[i] == p[j]	
actualizar	i = 11	j = 3



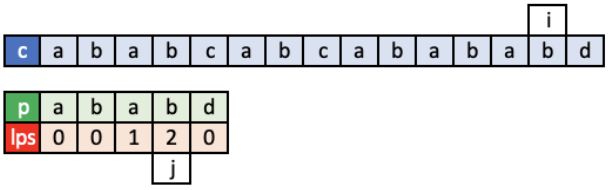
iteradores	i = 11	j = 3
caso	c[i] == p[j]	
actualizar	i = 12	j = 4



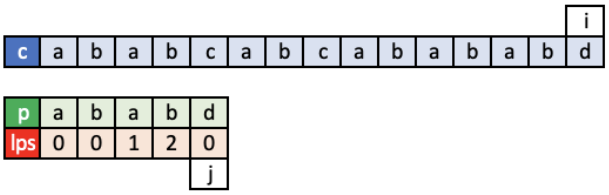
iteradores	i = 12	j = 4
caso	c[i] != p[j] && j != 0	
actualizar	j = lps[j-1] = lps[3] = 2	



iteradores	i = 12	j = 2
caso	c[i] == p[j]	
actualizar	i = 13	j = 3



iteradores	i = 13	j = 3
caso	c[i] == p[j]	
actualizar	i = 14	j = 4



iteradores	i = 14	j = 4
caso	c[i] == p[j] && j == p.size()	
actualizar	TRUE	

Como se puede observar, en 19 acciones pudimos determinar que el patrón es parte de la cadena, lo cual, considerando la longitud de ambos arreglos de caracteres, es óptimo.

Código

Complejidad de la solución: O(m+n)

```cpp
//Complejidad O(n)
vector<int> computeLPS(vector<char> pattern){
    vector<int> lps(pattern.size(), 0);

    int posPrevia = 0, posActual = 1;

    while (posActual < pattern.size()) {
        //Si dos posiciones consecutivas coinciden, significa que sus sufijos y prefijos también lo hacen,
        //por ello, lps en la posición actual es igual a la posición previa + 1. Movemos ambos iteradores
        //a la derecha
        if (pattern[posActual] == pattern[posPrevia]){
            lps[posActual] = posPrevia + 1;
            posPrevia += 1;
            posActual += 1;
        }
        //Si pp == 0, se traduce en que no hubo coincidencias, por lo que lps en
        //la posición actual es 0. Actualizamos el valor de pa+=1 (a la derecha)
        else if (posPrevia == 0){
            lps[posActual] = 0;
            posActual += 1;
        }
        //Si no coinciden recorremos el iterador pp a la izquierda, con respecto el valor de lps[pp-1], para
        //buscar alguna coincidencia. No necesariamente disminiye 1 unidad.
        else posPrevia = lps[posPrevia - 1];
    }

    return lps;
}
//Complejidad O(m+n)
int KMPSearch(vector<char> pat, vector<char> txt){
    int M = pat.size();                                  ⚠ Implicit conversion loses integer precision: 'std::vector<char>::size_ty
    int N = txt.size();                                  ⚠ Implicit conversion loses integer precision: 'std::vector<char>::size_ty

    //Valor para almacenar el índice en donde empieza la coincidencia en transmisiónX.txt
    int index = INT_MIN;

    //Mandamos llamar el preproceso para clacular la tabla LPS
    //Complejidad O(n)
    vector<int> lps = computeLPS(pat);

    //Iterador para la cadena (i) y subcadena (j)
    int i = 0, j = 0;

    //Complejidad O(m)
    while ((N - i) >= (M - j)) {
        //Hay coincidencia, recorremos a la derecha ambos iteradores
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }
        //Si j recorre toda la subcadena, significa que el patrón existe en el texto
        if (j == M) {
            //Calculamos el índice
            index = i - j;
            break;
            //j = lps[j - 1];
        }

        //Hay un mismatch
        else if (i < N && pat[j] != txt[i]) {
            // Nos apoyamos de la tabla LPS
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
    return index;
}
```

2) Palíndromo más largo

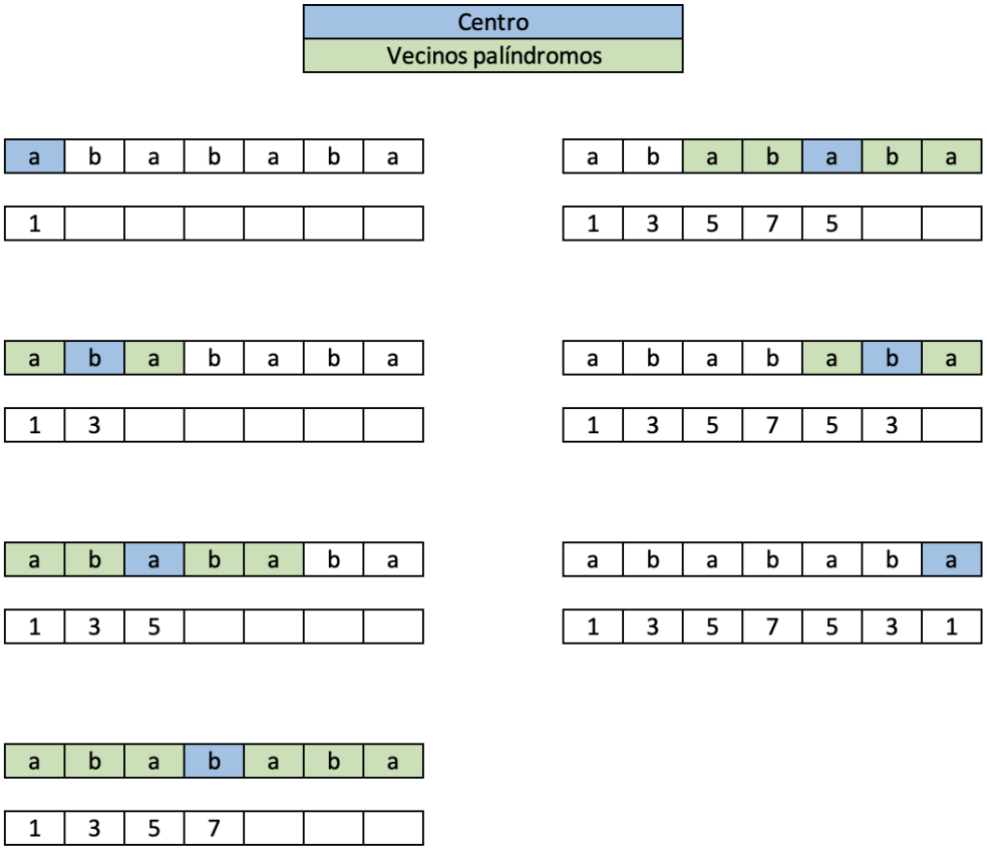
Un palíndromo es un conjunto de caracteres que se de igual manera, ya sea de izquierda a derecha, o bien, de derecha a izquierda. Ejemplos claros de este concepto son:

1. seres
2. solos
3. abba

Una manera de resolver este problema sería con la ideología de “expandir los centros”. El método consiste en:

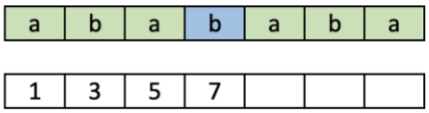
- Recorrer la cadena caracter por caracter, de izquierda a derecha.
- Cada caracter, cuando le corresponda, se vuelve el centro del palíndromo y comparará las letras (o valores alfanuméricos) de la izquierda y las de la derecha.
- Se cuenta la cantidad de letras que formen el palíndromo más largo con base en el centro designado y se almacenan los resultados en un arreglo de la misma longitud que la cadena.

Ejemplo:

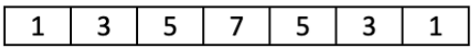


La complejidad de esta solución es $O(n^2)$ ya que tendríamos que programar 2 ciclos anidados, uno dedicado para recorrer la cadena y asignar los centros, y otro que servirá para corroborar que los valores vecinos son palíndromos. Es una solución poco eficiente. Por ello, resulta conveniente implementar otra solución: *Algoritmo de Manacher*.

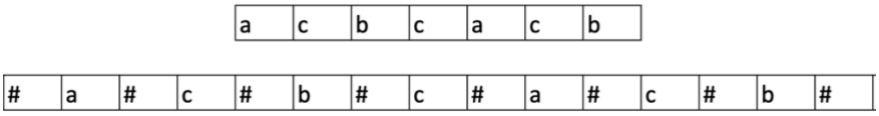
En el ejemplo previamente expuesto, hubo un momento en que llegamos a la mitad de la cadena y nos encontramos con lo siguiente:



Los valores a la izquierda del centro ya estaban calculados y se presentó un espejismo, por ello, ya no sería necesario calcular los valores a la derecha, simplemente sería cuestión de "copiar" los valores que ya tenemos. El resultado es el siguiente:



Cabe destacar que este método funciona solamente cuando el string es impar. Para un correcto funcionamiento de todos los casos se agregan gatos (1 a la izquierda y 1 a la derecha) a cada caracter de la cadena, además de caracteres especiales al inicio y al final del string:



Si el palíndromo de la mayor longitud se encuentra en una posición de un gato, sabemos que se toman los caracteres que envuelven al gato, solucionando el problema con los string de tamaño par.

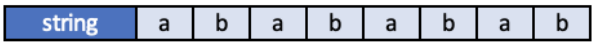
Consideraciones:

- Necesitamos un arreglo *len* que almacene la longitud de todos los palíndromos
- *i* = Índice. Inicia en 1.
- *c* = Centro del último palíndromo con el que estemos trabajando. Inicia en 0.
- *r* = Límite derecho del último palíndromo con el que estemos trabajando. Inicia en 0.

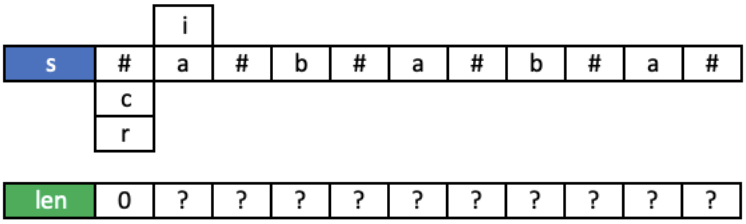
Funcionamiento:

- Si la posición actual es menor a la barrera de la derecha (i < r), podemos utilizar su valor "espejo" para actualizar su palíndromo (longitud = min (*r*-*i*, espejo)).
- Después, hacemos la expansión a partir del centro para encontrar el tamaño del palíndromo
- Finalmente, verificamos si el palíndromo actual se expande más allá de la barrera de la derecha, necesitamos reasignar el centro (*c* = *i*, *r* = *i* + *len*[*i*]).

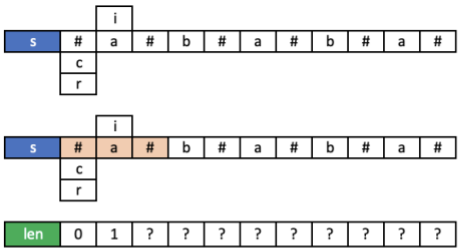
A continuación, se mostrará un ejemplo del funcionamiento de este algoritmo.
Consideremos el siguiente string:



Lo primero que debemos hacer es colocar los gatos en las posiciones correspondientes y generar el vector que almacenará las longitudes de los palíndromos:

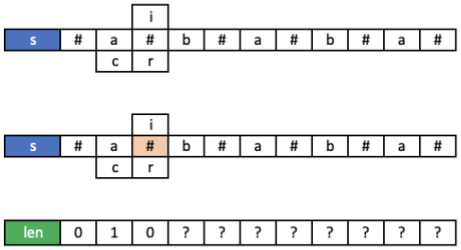


Considerando las 3 reglas previamente descritas, así como la posición de los iteradores, podemos determinar que:



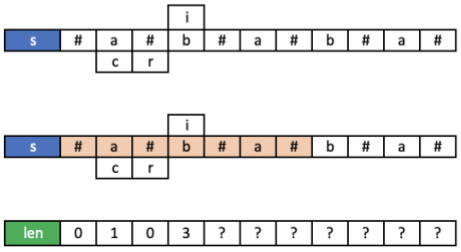
iteradores	i = 1	c = 0	r = 0

1) if (i<r) --> longitug = min(r-i, espejo)	1<0	false	
2) expand	1		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	1+1>0	true	c=1, r=1+1



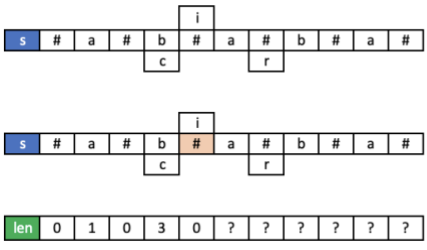
iteradores	i = 2	c = 1	r = 2

1) if (i<r) --> longitug = min(r-i, espejo)	2<2	false	
2) expand	0		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	2+0>2	false	



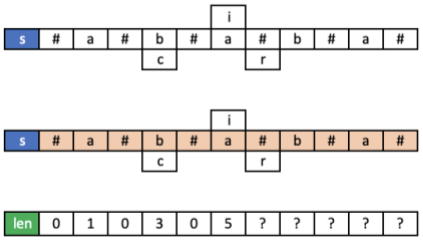
iteradores	i = 3	c = 1	r = 2

1) if (i<r) --> longitug = min(r-i, espejo)	3<2	false	
2) expand	3		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	3+3>2	true	c=3, r=3+3



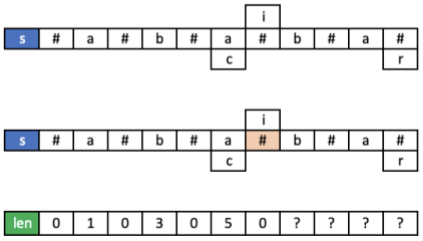
iteradores	i = 4	c = 3	r = 6

1) if (i<r) --> longitug = min(r-i, espejo)	4<6	true	len[i] = min(6-1, 0)
2) expand	0		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	4+0>6	false	



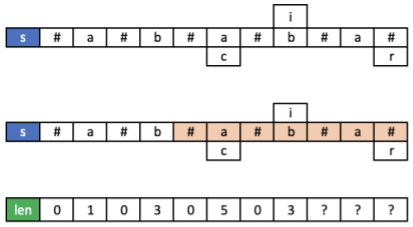
iteradores	i = 5	c = 3	r = 6

1) if (i<r) --> longitug = min(r-i, espejo)	5<6	true	len[i] = min(6-5, 1)
2) expand	5		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	5+5>6	true	c=5, r=5+5



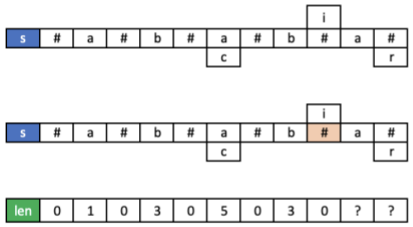
iteradores	i = 6	c = 5	r = 10

1) if (i<r) --> longitug = min(r-i, espejo)	6<10	true	len[i] = min(10-6, 0)
2) expand	0		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	6+0>10	false	



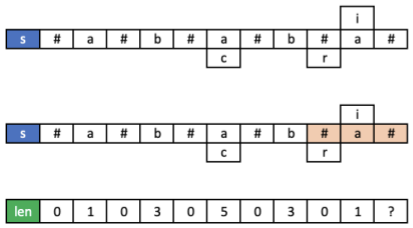
iteradores	i = 7	c = 5	r = 10

1) if (i<r) --> longitug = min(r-i, espejo)	7<10	true	len[i] = min(10-7, 0)
2) expand	3		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	7+3>10	false	



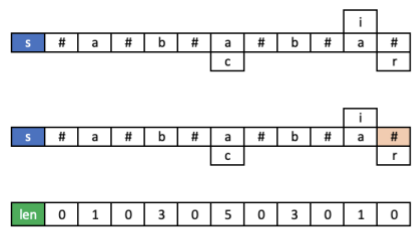
iteradores	i = 8	c = 5	r = 10

1) if (i<r) --> longitug = min(r-i, espejo)	7<10	true	len[i] = min(10-8, 0)
2) expand	0		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	8+3>10	true	c=8, r=8+0



iteradores	i = 9	c = 8	r = 8

1) if (i<r) --> longitug = min(r-i, espejo)	9>8	false	
2) expand	1		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	9+1>8	true	c=9, r=9+1



iteradores	i = 10	c = 9	r = 10

1) if (i<r) --> longitug = min(r-i, espejo)	10>10	false	
2) expand	0		
3) if (i+len[i]>r) --> center = i, r = i+len[i]	10+0>10	false	

Longest Palindrome = 5

Código

Complejidad de la solución: O(n)

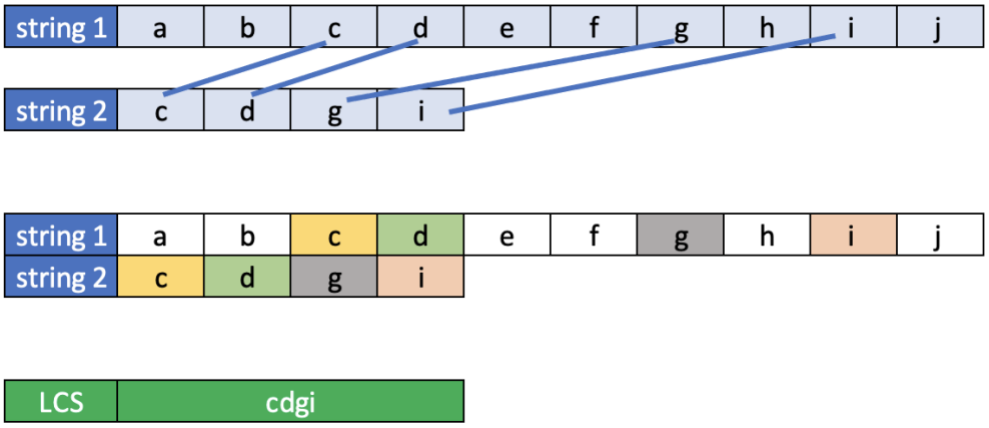
```
//Complejidad O(n)
pair<int, int> longestPalindrome(vector<char> cadena) {
    //Creamos un vector char que contenga los caracteres especiales
    vector<char> cadManacher;

    //Complejidad O(n)
    for (int i = 0; i < cadena.size(); i++){
        cadManacher.push_back('#');
        cadManacher.push_back(cadena[i]);
    }
    cadManacher.push_back('#');

    //Creamos el vector que almacene las diferentes longitudes de los palíndromos
    vector<int> longPalindrome(cadManacher.size(), 0);

    //Definimos 2 variables que funcionarán como apuntadores:
    //centro (línea de espejismo) y límite derecho
    int center = 0, right = 0;

    //Iteramos cadManacher de manera efectiva, es decir, evitando,
    //estratégicamente, algunas iteraciones
    //Complejidad O(n)
    for (int i = 1; i < longPalindrome.size() - 1; i++) {
        //Valor del índice a la izquierda del espejo
        int mirror = center - (i - center);

        //Corroborar que el iterador i no sobrepasa el límite de la derecha (right)
        if (right > i) {
            longPalindrome[i] = min(right  - i, longPalindrome[mirror]);
        }

        //Aplicamos la "expansión"
        while (cadManacher[i + (1 + longPalindrome[i])] == cadManacher[i - (1 + longPalindrome[i])]) {
            longPalindrome[i]++;
        }

        // Si la expansión va más allá del límite derecho, es necesario actualizar el centro y el límite derecho
        if (i + longPalindrome[i] > right) {
            center = i;
            right = i + longPalindrome[i];
        }
    }
    //Buscamos el palíndromo más largo y su índice
    int maxLen = 0, centerIdx = 0;
    for (int i = 1; i < longPalindrome.size() - 1; i++) {
        if (longPalindrome[i] > maxLen) {
            maxLen = longPalindrome[i];
            centerIdx = i;
        }
    }

    int inicio =(centerIdx - 1 - maxLen) / 2;
    int fin = (centerIdx - 1 + maxLen) / 2;

    //El índice debe iniciar en 1, no en 0
    pair<int, int> res = make_pair(inicio + 1, fin + 1);
    return res;
}
```

3) Subsecuencia más largoa en común

Cuando hablamos de la subsecuencia más larga (Longest Common Subsequence, LCS) nos referimos a, valga la redundancia, la subsecuencia más larga que es común a todas las secuencias dadas, siempre que no se requiera que los elementos de la subsecuencia ocupen posiciones consecutivas dentro de las secuencias originales.

En LCS, a diferencia de los 2 puntos anteriores, recibes 2 strings completos, es decir, no recibes una cadena y una subcadena. Generalmente, al string más grande se le atribuye la primera posición en el código y la segunda posición a la otra cadena.

Ejemplo:



Como se puede observar, los caracteres, no necesariamente, están en locaciones consecutivas, sin embargo, si aparecen en el mismo orden.

Existen diversos métodos para resolver este inciso:

- Con recursión
- Con memoización
- Con programación dinámica

Para fines de este proyecto, utilizaremos la programación dinámica.

Consideraciones:

- Necesitamos 2 strings: x y y.
- Creamos una tabla de tamaño (x.size()+1)*(y.size()+1).
- La primera fila, así como la primera columna se iniciarán con “*” debido a que no es necesario asignarle un valor. Su función es permitir recorrer la tabla en su totalidad.

Funcionamiento:
Caso 1:
Caso 2:
Caso 3:
Ejemplo:

Considerando la siguiente información, mostraremos como se resuelve este problema:

string 1	a	c	a	d	b

string 2	c	b	d	a

		c	b	d	a
	*	*	*	*	*
a	*				
c	*				
a	*				
d	*				
b	*				

		c	b	d	a
	0	0	0	0	0
a	0				
c	0				
a	0				
d	0				
b	0				

		c	b	d	a
	*	*	*	*	*
a	*	L	L	L	D
c	*				
a	*				
d	*				
b	*				

		c	b	d	a
	0	0	0	0	0
a	0	0	0	0	1
c	0				
a	0				
d	0				
b	0				

iteradores	i = 1	j = 1	i = 1	j = 2	i = 1	j = 3	i = 1	j = 4
condición	C[i-1][j] < C[i][j-1]		C[i-1][j] < C[i][j-1]		C[i-1][j] < C[i][j-1]		x[i-1] == y[j-1]	
caso	C[i][j] = C[i][j-1] = 0		C[i][j] = C[i][j-1] = 0		C[i][j] = C[i][j-1] = 0		C[i-1][j-1] + 1 = 1	
caso	L		L		L		D	

		c	b	d	a
	*	*	*	*	*
a	*	L	L	L	D
c	*	D	L	L	L
a	*				
d	*				
b	*				

		c	b	d	a
	0	0	0	0	0
a	0	0	0	0	1
c	0	1	1	1	1
a	0				
d	0				
b	0				

iteradores	i = 2	j = 1	i = 2	j = 2	i = 2	j = 3	i = 2	j = 4
condición	x[i-1] == y[j-1]		C[i-1][j] < C[i][j-1]		C[i-1][j] < C[i][j-1]		C[i-1][j] < C[i][j-1]	
caso	C[i-1][j-1] + 1 = 1		C[i][j] = C[i][j-1] = 1		C[i][j] = C[i][j-1] = 1		C[i][j] = C[i][j-1] = 1	
caso	D		L		L		L	

		c	b	d	a
	*	*	*	*	*
a	*	L	L	L	D
c	*	D	L	L	L
a	*	U	L	L	D
d	*				
b	*				

		c	b	d	a
	0	0	0	0	0
a	0	0	0	0	1
c	0	1	1	1	1
a	0	1	1	1	2
d	0				
b	0				

iteradores	i = 3	j = 1	i = 3	j = 2	i = 3	j = 3	i = 3	j = 4
condición	C[i-1][j] > C[i][j-1]		C[i-1][j] < C[i][j-1]		C[i-1][j] < C[i][j-1]		x[i-1] == y[j-1]	
caso	C[i][j] = C[i-1][j] = 1		C[i][j] = C[i][j-1] = 1		C[i][j] = C[i][j-1] = 1		C[i-1][j-1] + 1 = 2	
caso	U		L		L		D	

		c	b	d	a
	*	*	*	*	*
a	*	L	L	L	D
c	*	D	L	L	L
a	*	U	L	L	D
d	*	U	L	D	L
b	*				

		c	b	d	a
	0	0	0	0	0
a	0	0	0	0	1
c	0	1	1	1	1
a	0	1	1	1	2
d	0	1	1	2	2
b	0				

iteradores	i = 4	j = 1	i = 4	j = 2	i = 4	j = 3	i = 4	j = 4
condición	C[i-1][j] > C[i][j-1]		C[i-1][j] < C[i][j-1]		x[i-1] == y[j-1]		C[i-1][j] < C[i][j-1]	
caso	C[i][j] = C[i-1][j] = 1		C[i][j] = C[i][j-1] = 1		C[i-1][j-1] + 1 = 2		C[i-1][j-1] + 1 = 2	
caso	U		L		D		L	

		c	b	d	a
	*	*	*	*	*
a	*	L	L	L	D
c	*	D	L	L	L
a	*	U	L	L	D
d	*	U	L	D	L
b	*	U	D	L	L

		c	b	d	a
	0	0	0	0	0
a	0	0	0	0	1
c	0	1	1	1	1
a	0	1	1	1	2
d	0	1	1	2	2
b	0	1	2	2	2

iteradores	i = 5	j = 1	i = 5	j = 2	i = 5	j = 3	i = 5	j = 4
condición	C[i-1][j] > C[i][j-1]		x[i-1] == y[j-1]		C[i-1][j] < C[i][j-1]		C[i-1][j] < C[i][j-1]	
caso	C[i][j] = C[i-1][j] = 1		C[i-1][j-1] + 1 = 2		C[i][j] = C[i][j-1] = 2		C[i][j] = C[i][j-1] = 2	
caso	U		D		L		L	

Código

Complejidad de la solución: O(m*n) - Optimizado

```cpp
//Complejidad O(n*m)
//Como solo necesitamos calcular los índices, podemos evitar pasos para acelerar la función
//pair<int, int> makeLCSTables(vector<char> x, vector<char> y, vector<vector<string>> &B, vector<vector<int>> &C){
pair<int, int> makeLCSTables(vector<char> x, vector<char> y){
    int m = x.size();
    int n = y.size();

    //Variabales para una bandera, el índice de inicio y el índice final
    int flag = 0, start = 0, end = 0;

    /*
    for (int i = 0; i <= m; i++){
        vector<int> rowC(n+1, 0);
        C.push_back(rowC);

        vector<string> rowB(n+1, "*");
        B.push_back(rowB);
    }
     */
    //Complejidad O(m*n)
    for (int i = 1; i <= m; i++){
        for (int j = 1; j <= n; j++){
            //Encontramos una coincidencia, por ello, almacenamos el índice en la variable
            //que corresponda
            if (x[i-1] == y [j-1]){
                //C[i][j] = C[i-1][j-1] + 1;
                //B[i][j] = "D";
                //Si flag == 0, significa que i representa el índice del inicio
                 if (flag == 0){
                     start = i;
                     flag = 1;
                 }
                 //En caso contrario, actualizamos el valor del índice que representa el final
                 else end = i;
             }
             else{
                 /*
                 if (C[i-1][j] > C[i][j-1]){
                     C[i][j] = C[i-1][j];
                     B[i][j] = "U";
                 }
                 else{
                     C[i][j] = C[i][j-1];
                     B[i][j] = "L";
                 }
                  */
                 continue;
             }
         }
    }

    pair<int, int> res = make_pair(start, end);
    return res;
}
```

Implicit conversion loses integer precision: 'std::vector<char>::size_type' (a

Implicit conversion loses integer precision: 'std::vector<char>::size_type' (a

Resultados

Para calcular la solución, es necesario crear una función que pueda leer archivos y, además, vuelva al string en un vector de tipo char:

```
vector<char> genCharVectorFromString(string s){
    vector<char> vecChar(s.begin(), s.end());
    return vecChar;
}

vector<char> readFile(string s){
    ifstream lector(s);
    if (lector.fail()){
        cout << "\nError en la carga del archivo" << endl;
        exit(1);
    }
    string str((istreambuf_iterator<char>(lector)), istreambuf_iterator<char>());
    return genCharVectorFromString(str);
}
```

Los resultados obtenidos fueron los siguientes:

```
PARTE 1: Substring Search (KMP Algorithm)
    transmission1.txt && mcode1.txt --> true   |   index: 2241
    transmission1.txt && mcode2.txt --> true   |   index: 3370
    transmission1.txt && mcode3.txt --> true   |   index: 5061
    transmission2.txt && mcode1.txt --> false
    transmission2.txt && mcode2.txt --> false
    transmission2.txt && mcode3.txt --> false

PARTE 2: Longest Palindromic String (Manacher Algorithm)
    LPS transmission1.txt --> startIndex: 5061  |  lastIndex: 5106
    LPS transmission2.txt --> startIndex: 1995  |  lastIndex: 2003

PARTE 3: Longest Common Subsequence (Dynamic Programming)
    LCS transmission1.txt --> startIndex: 1  |  lastIndex: 5755
    LCS transmission2.txt --> startIndex: 1  |  lastIndex: 5551

Program ended with exit code: 0
```

Conclusiones

En lo personal, considero que esta actividad integradora ha sido una de las más complejas a las que me he enfrentado, ya que para llegar a una solución fue necesario asimilar el problema, así como estudiar detalladamente el funcionamiento y lógica de cada uno de los algoritmos implementados en esta actividad. De igual manera, pienso que una de las estrategias que nos ayudaron a comprender mejor los algoritmos fue el elaborar manual y visualmente la ejecución paso a paso de estos. Gracias a esto, considero que pudimos lograr de forma óptima la implementación de nuestro programa, ya que los algoritmos desarrollados resultaron ser eficientes al tener una complejidad no tan elevada.

De igual manera, esta actividad me ayudó a reflexionar sobre lo importante que es el diseñar e implementar algoritmos que sean confiables y eficientes, puesto que los recursos de una computadora son limitados, por lo que el optimizar algoritmos computacionales

nos permiten sacar mayor provecho de estos para aplicarlos en el desarrollo de mejores soluciones.

Referencias

Bari, A. [UCZCFT11CWBi3MHNlGf019nw]. (2018, marzo 25). *9.1 Knuth-Morris-Pratt KMP string matching algorithm*. Youtube. https://www.youtube.com/watch?v=V5-7GzOfADQ

KMP algorithm for pattern searching. (2011, abril 3). GeeksforGeeks. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

Loading. (s/f). Leetcode.com. Recuperado el 9 de septiembre de 2022, de https://leetcode.com/problems/longest-palindromic-substring/discuss/378722/Java-Manacher's-Algorithm-with-detailed-comments-beats-99.34!

Longest common subsequence. (s/f). Programiz.com. Recuperado el 9 de septiembre de 2022, de https://www.programiz.com/dsa/longest-common-subsequence

Time Complexity Infinity [UCf9ywn38ypqEIC6McbwH9sQ]. (2018, diciembre 17). LeetCode 5 - Longest Palindromic Subtring. Youtube. https://www.youtube.com/watch?v=SV1ZaKCozS4