# PARTE 1

String = **“Esto es un string”** (usamos string the python)

## Ejercicio 1 (opcional)

Implementar la función que responde a la siguiente especificación.

**def existChar(String, c):**

**Descripción:** Confirma la existencia de un carácter específico en una cadena.

**Entrada: String** con la cadena en la cual buscar el carácter, **carácter** a buscar en la cadena.

**Salida:** Retorna **True** si el carácter se encuentra en la cadena, o **False** en caso contrario

## Ejercicio 2 (opcional)

Implementar una función que detecte si una cadena es un Palíndromo. La implementación debe responder a la siguiente especificación:

**def isPalindrome(String):**

**Descripción:** Determina si la cadena es un palíndromo

**Entrada: String** con la cadena a evaluar.

**Salida:** Retorna **True** si la cadena es palíndromo, o **False** en caso contrario

La función es Palíndromo que devuelve True si una cadena es Palindromo y Falso en caso contrario. Nota: Una cadena es un palíndromo si se lee igual en ambos sentidos ej. anitalavalatina, radar.

## Ejercicio 3 (opcional)

Implementar la función que responde a la siguiente especificación.

**def mostRepeatedChar(String):**

**Descripción:** Encuentra el carácter que más se repite en una cadena.

**Entrada: String** con la cadena a ser evaluada.

**Salida:** Retorna el carácter que más se repite. En caso que haya más de un carácter con mayor ocurrencia devuelve el primero de ellos.

## Ejercicio 4 (opcional)

Implementar la función que dado un String S devuelve la longitud de la isla de mayor tamaño. Una isla es una secuencia consecutiva de un mismo carácter dentro de S. Por

ejemplo S = **“cdaaaaaasssbbb”** su mayor isla es de tamaño 6 (aaaaaa) y además tiene dos islas de tamaño 3 (sss, bbb) el resto de las islas en s son de tamaño 1.

**def getBiggestIslandLen(String):**

**Descripción:** Determina el tamaño de la isla de mayor tamaño en una cadena.

**Entrada: String** con la cadena a ser evaluada.

**Salida:** Retorna un **entero** con la dimensión de la isla más grande dentro de la cadena.

# Ejercicio 5 (opcional)

Implementar la función que responde a la siguiente especificación.

**def isAnagram(String, String):**

**Descripción:** Determina si una cadena es un anagrama de otra.

**Entrada:** Un **String** con la cadena original, y otro **String** con el posible anagrama a evaluar.

**Salida:** Retorna un **True** si la segunda cadena es anagrama de la primera, en caso contrario devuelve **False.**

Nota: Una cadena **s** es anagrama de otra cadena **p** si existe alguna ordenación de los elementos de **s** con lo cual se obtenga la cadena **p**

# Ejercicio 6 (opcional)

Implementar la función que responde a la siguiente especificación.

**def verifyBalancedParentheses(String):**

**Descripción:** Verifica si los paréntesis contenidos en una cadena se encuentran balanceados y en orden.

**Entrada:** Un **String** con la cadena a ser evaluada.

**Salida:** Retorna un **True** si la cadena posee sus paréntesis correctamente balanceados, en caso contrario devuelve **False.**

Ejemplo: **“(ccc(ccc)cc((ccc(c))))”** es correcto, pero **“)ccc(ccc)cc((ccc(c)))(“** no lo es, aunque tenga el mismo número de paréntesis abiertos que cerrados.

# Ejercicio 7

Se tiene una cadena de caracteres y se quiere reducir a su longitud haciendo una serie de operaciones. En cada operación se selecciona **un par** de caracteres adyacentes que coinciden, y se los borra. Por ejemplo, la cadena **“aab”** puede ser acortada a **“b”** en una sola operación. Implementar una función que borre tantos caracteres como sea posible y devuelva la cadena resultante.

**def reduceLen(String):**

**Descripción:** Reduce la longitud de una cadena removiendo iterativamente pares de caracteres repetidos.
**Entrada:** Un **String** con la cadena a ser reducida.
**Salida:** Retorna un **String** con la cadena resultante tras haber aplicado las remociones.

Ejemplo: **“aaabccddd”** se puede reducir a **“abd”** de la siguiente manera:
**“aaabccddd”** → **“abccddd”** → **“abddd”** → **“abd”**

```python
#--EJERCICIO 7--
def reduceLen(string):
    pila = []

    for char in string:
        if pila and pila[-1] == char:
            pila.pop()  # Remueve el último elemento del stack si es igual al actual
        else:
            pila.append(char)  # Agrega el caracter actual al stack

    return ''.join(pila)  # Convierte la lista a una cadena
```

## Ejercicio 8

Implementar una función que dadas dos palabras determine si la segunda está contenida dentro de la primera bajo la siguiente premisa. Una cadena s contiene la palabra **“amarillo”** si un subconjunto ordenado de sus caracteres deletrea la palabra **amarillo**. Por ejemplo, la cadena **s = "aaafffmmmarillzzzllhooo"** contiene **amarillo**, pero **s = "aaafffmmmarrrilzzzhooo"** no (debido a que le falta una **l**). Si ordenamos la primera cadena como **s = "aaaaillllfffzzzhrmmmooo"**, ya no contiene la subsecuencia debido al ordenamiento.

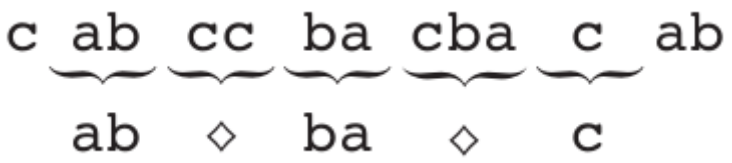
**def isContained(String,String):**

**Descripción:** Determina si los caracteres de una cadena se encuentran contenidos y en el mismo orden dentro de otra cadena.
**Entrada:** Un **String** con la cadena a evaluar, y otro **String** con la cadena posiblemente contenida en la primera.
**Salida:** Retorna un **True** si la segunda cadena se encuentra contenida en la primera, o **False** en caso contrario.

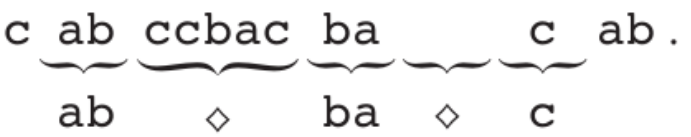
```python
#--EJERCICIO 8--
def isContained(main_str, sub_str):
    m, n = len(main_str), len(sub_str)
    i, j = 0, 0

    # Recorre los caracteres de main_str
    while i < m and j < n:
        if main_str[i] == sub_str[j]:
            j += 1  # Avanza el puntero de sub_str
        i += 1  # Avanza el puntero de main_str

    # Si hemos recorrido toda la sub_str, significa que
    return j == n

```

## Ejercicio 9

Suponga que se quiere encontrar si existe la ocurrencia exacta de una cadena **p** dentro de una cadena **s.** Suponga que se permite que el patrón tenga caracteres comodín que pueden matchear con cualquier cadena de caracteres (incluso de longitud 0). Por ejemplo, el patrón **"ab◇ba◇c"** ocurre en el texto **"cabccbacbacab"** como sigue:




Y también como




Note que el carácter comodín (◇) puede aparecer un número arbitrario de veces en el patrón **p**, pero se asume que no aparecerá en la cadena **s**. Proponga un algoritmo en tiempo polinomial para determinar si un patrón **p** aparece en un texto **s** dado.

**def isPatternContained(String,String,c):**

**Descripción:** Determina en tiempo polinomial si un patrón de caracteres conformado por caracteres fijos y comodines se encuentra en otra cadena.

**Entrada:** Un **String** con la cadena a evaluar, un **String** con el patrón a buscar, y un carácter **c** que especifica el carácter comodín dentro del patrón.

**Salida:** Retorna un **True** si el patrón proporcionado se encuentra en la cadena, o **False** en caso contrario.

# PARTE 2

## Ejercicio 10

Construir un Autómata de Estados Finitos para el patrón **P="aabab"** y demostrar su funcionamiento en la cadena de texto **T="aaababaabaababaab"**. **No es necesario implementar.**

| state | a | b | P: |
|---|---|---|---|
| 0 | 1 | 0 | a |
| 1 | 2 | 0 | a |
| 2 | 1 | 3 | b |
| 3 | 4 | 0 | a |
| 4 | 1 | 5 | b |

## Ejercicio 11

Sean el texto T y el patrón P de longitudes m y n respectivamente. Plantee un algoritmo para encontrar el mayor prefijo de P que se encuentra en T en **O(n+m)**.

## Ejercicio 12

Implementar en pseudo-python un autómata de estados finitos para buscar cualquier patrón P (consecutivo) en una cadena de texto T.

```python
#--EJERCICIO 12--
def construirAEF(patron):
    m = len(patron)
    aef = [{} for _ in range(m + 1)] #lista con diccionarios {caracter: estado}}

    # defino el estado inicial
    aef[0][patron[0]] = 1
    X = 0  # Estado de fallo

    for j in range(1, m):
        for c in set(patron):
            aef[j][c] = aef[X].get(c, 0)
        aef[j][patron[j]] = j + 1
        X = aef[X].get(patron[j], 0)

    for c in set(patron):
        aef[m][c] = aef[X].get(c, 0)

    return aef
                                                    Generate Ctrl
def buscar_patron(patron, texto):
    automata = construirAEF(patron)
    m, n = len(patron), len(texto)
    i, j = 0, 0  # i para texto, j para patrón (estado del DFA)

    while i < n:
        j = automata[j].get(texto[i], 0)
        if j == m:
            print(f"Patrón encontrado en la posición {i - m + 1}")
            j = 0  # Reiniciar para buscar más coincidencias
        i += 1
```

# Ejercicio 13

Implemente el algoritmo de Rabin-Karp estudiado. Para el mismo deberá implementarse una función de hash que dado un patrón p de tamaño m se resuelva en O(1). Considerar lo detallando en las presentación del tema correspondiente a las funciones de hash en Rabin-karp.

```python
def rabin_karp(p, t, q):
    m = len(p)  # Longitud del patrón
    n = len(t)  # Longitud del texto
    d = 256  # Número de caracteres en el alfabeto (por ejemplo, ASCII tiene 256 caracteres)
    hp = 0  # Hash del patrón
    ht = 0  # Hash del texto
    h = 1  # Valor de h usado en el cálculo del hash

    # Calcular h = d^(m-1) % q
    for i in range(m - 1):
        h = (h * d) % q

    # Calcular el hash inicial del patrón y del primer segmento del texto
    for i in range(m):
        hp = (d * hp + ord(p[i])) % q
        ht = (d * ht + ord(t[i])) % q

    # Buscar el patrón en el texto
    for i in range(n - m + 1):
        # Si los hashes coinciden, verificar carácter por carácter
        if hp == ht:
            match = True
            for j in range(m):
                if t[i + j] != p[j]:
                    match = False
                    break
            if match:
                print(f"{p} encontrado en la posición {i + 1}")
```

```python
    # Actualizar el hash para la siguiente ventana de texto
    if i < n - m:
        ht = (d * (ht - ord(t[i]) * h) + ord(t[i + m])) % q
        if ht < 0:
            ht += q

# Si no se encontró ninguna coincidencia
print("Ninguna coincidencia encontrada")
```

# Ejercicio 14

Implemente el algoritmo KMP estudiado.

**def KMP(String,String):**

**Descripción:** Implementa el algoritmo KMP.

**Entrada:** Un **String** con la cadena a evaluar, y un **String** con el patrón a buscar.

**Salida:** Retorna un arreglo de índices con las posiciones en donde se encuentra el patrón, o **None** en caso de no encontrar el patrón.

```
102  #--EJERCICIO 14--
103  def KMP(t, p):
104      n = len(t)  # Longitud del texto
105      m = len(p)  # Longitud del patrón
106      pi = computePrefixFunction(p)  # Calcular la función de prefijo del patrón
107      q = 0  # Índice que indica cuántos caracteres coinciden
108
109      for i in range(0, n):
110          # Mientras exista un mismatch y q sea mayor que 0, retroceder según la
     función de prefijo
111          while q > 0 and p[q] != t[i]:
112              q = pi[q - 1]
113
114          # Si el caracter actual del texto coincide con el caracter actual del
     patrón
115          if p[q] == t[i]:
116              q += 1  # Incrementar q, indicando un nuevo caracter coincidente
117
118          # Si q es igual a la longitud del patrón m, se encontró una ocurrencia
     completa
119          if q == m:
120              print("Pattern occurs with shift", i - m + 2)
121              break
122
123      # Si q no es igual a m, significa que no se encontraron ocurrencias
124      if q != m:
125          print("None")
```

```
127  def computePrefixFunction(p):
128      m = len(p)  # Longitud del patrón
129      pi = [0] * m  # Inicializar arreglo para la función de prefijo
130      k = 0  # Longitud del prefijo más largo que también es un sufijo
131
132      for q in range(1, m):
133          # Mientras exista un mismatch, retroceder según la función de prefijo
134          while k > 0 and p[k] != p[q]:
135              k = pi[k - 1]
136
137          # Si el caracter siguiente coincide, incrementar k
138          if p[k] == p[q]:
139              k += 1
140
141          pi[q] = k  # Asignar el valor de k a la posición q de la función de
     prefijo
142
143      return pi  # Retornar la función de prefijo computada
144
```

# Ejercicio 15 (opcional)

Realice una modificación al algoritmo KMP para encontrar las ocurrencias no solapadas del patrón P en el texto T. Por ejemplo: si P = aba y T = aababababaaa las ocurrencias de P a**aba**bababaaa y aab**aba**baaa se solapan por lo que la mayor cantidad de ocurrencias no solapadas son 2, o sea a**aba**b**aba**aa.

**def KMPmod(String,String):**

**Descripción:** Implementa el algoritmo KMP sin solapado.

**Entrada:** Un **String** con la cadena a evaluar, y un **String** con el patrón a buscar.

**Salida:** Retorna un arreglo de índices con las posiciones en donde se encuentra el patrón sin solapado, o **None** en caso de no encontrar el patrón.

A tener en cuenta:

1. **Usen lápiz y papel primero**
2. **~~No se puede utilizar otra Biblioteca mas allá de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.~~**