NAHMAN MARTINA

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"**

```
def reduce_len(string):
    string = str(string)  #convierte el string si no es
    new_string = ""
    flag=False
    for i in range(len(string)):
        if flag == False:
            if i == len(string)-1 or string[i] != string[i+1]:
                new_string += string[i]
            else:
                flag = True
        else:
            flag = False
    return new_string
```

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.

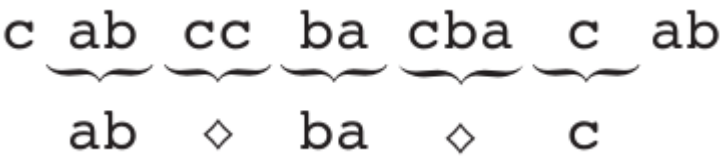
Entrada: Un **String** con la cadena a evaluar, y otro **String** con la cadena posiblemente contenida en la primera.

Salida: Retorna un **True** si la segunda cadena se encuentra contenida en la primera, o **False** en caso contrario.

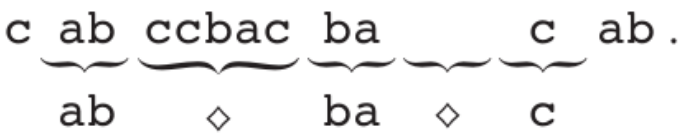
```
#Ejerciico 8
def  isContained(string1,string2):
    palabra = sorted(string1)  #ordena la primer entrada
    indice = 0
    string2 = list(string2)
    for char in palabra:
        if char == string2[indice]:
            indice += 1  #recorre
            if indice == len(string2):
                return True  #encontrada
    return False
```

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.

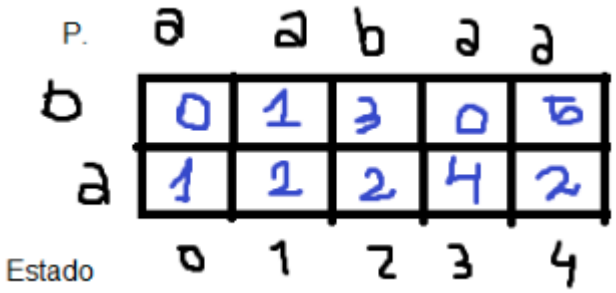
Entrada: Un **String** con la cadena a evaluar, un **String** con el patrón a buscar, y un carácter **c** que especifica el carácter comodín dentro del patrón.

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.**



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)**.

```
def prefijo_mayor(T, P):
    m = len(T)
    n = len(P)
    i = 0
    j = 0
    pre_mayor = 0  #len(del mayor prefijo encontrado)
    flag=True
    while i < m and j < n and flag==True:
        if T[i] == P[j]:
            i += 1
            j += 1
            pre_mayor += 1
        else:
            flag=False
    return T[:pre_mayor]
```

Ejercicio 12

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

```
def finite_automaton_matcher(t,p):
    n=len(t)
    q=0
    matriz= transition_function(t, p)
    for i in range (n):
        q=matriz[q,t[i]]
        if q==len(p):
            print('patter occurs withshift',i-len(p))
```

```python
def transition_function(p, alf):
    m = len(p)
    filas = m
    columnas = len(alf)
    valor_inicial = 0
    matriz = [[valor_inicial] * columnas for _ in range(filas)]
    for i in range(m):
        for j in range(len(alf)):
            k = min(m + 1, i + 2)
            k =k- 1
            while sufijo(p[0:k], (p + j)[0:i]):
                matriz[i][j] = k
    return matriz
```

```python
def sufijo(string1, string2):
   return string2[len(string2) - len(string1):len(string2)] == string1
```

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 RK(s,p):
    for i in range (len(s)-len(p)+1): #+1 para que tome el ultimo
        sub_s= s[i:len(p)+i] #nos devolvera las subcadenas de s con longitud de p
        if hash(sub_s) == hash(p):
            if sub_s==p:
                return True
    return False

def hash(string):
   unique_num = 0
   index_pow = 0
   for i in range(len(string)-1, 0, -1):
      unique_num += pow(128, index_pow) * ord(string[i])
      index_pow += 1
   return unique_num % len(string)
```

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.

```python
def pi_function(string):
    m=len(string)
    pi=[0]*len(string)
    pi[0]=0
    i=0
    for q in range(1,m,1):
        while i>0 and string[i] != string[q]:
            i=pi[i]
        if string[i] == string[q]:
            i=i+1
        pi[q]=i
    return pi
```

```python
def KMP(t,p):
    n=len(t)
    m=len(p)
    pi=pi_function(p)
    i=0
    for j in range(1,n,1):
        while i>0 and p[i]!=t[j]:
            i=pi[i]
        if p[i]==t[j]:
            i=i+1
        if i==m:
            return True
    return False
```

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**babaaa 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.~~**