

## 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.

```
16 def isPalindrome(s):
17     long = len(s)
18     palindrome = True
19     if len(s) % 2 != 0:
20         r = math.trunc((long-1)/2)
21         long = long-1
22         for i in range(0,r):
23             if s[i] != s[long]:
24                 palindrome = False
25                 break
26             long -= 1
27         return palindrome
28
```

### 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”

## 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** = “aaaffmmmarillzzllhooo” contiene **amarillo**, pero **s** = “aaaffmmmmarrilzzzhooo” no (debido a que le falta una l). Si ordenamos la primera cadena como **s** = “aaaaiilllfffzzzhrrmmmmooo”, 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.

## 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 “cabccbaccbacab” como sigue:

c   ab   cc   ba   cba   c   ab  
    └──┘ └──┘ └──┘ └──┘ └──┘  
    ab   ♦   ba   ♦   c

Y también como

c ab ccbac ba c ab .  
ab     ◇     ba     ◇     c

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.

```
88 def construirAutomata(patron,m):
89
90     automata = [[0] * 128 for _ in range(m + 1)] #tabla de estados
91     automata[0][ord(patron[0])] = 1 #paso inicial: primera transición
92     x = 0 #referencia para las transiciones cuando se encuentra un carácter
93     for estado in range(1, m + 1): #para recorrer el patron en todo
94         for c in range(128):
95             automata[estado][c] = automata[x][c]
96             if estado < m:
97                 automata[estado][ord(patron[estado])] = estado + 1
98                 #Si el autómata se encuentra en el estado 'estado' y se encuentra el carácter 'c'
99                 x = automata[x][ord(patron[estado])]
100             #Esto garantiza que si se encuentra un carácter que no está en el patrón, se mantenga el estado anterior
101
102     return automata
```

## 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 presentaciones del tema correspondiente a las funciones de hash en Rabin-karp.

```
32
33 def RK(p, t, q):
34     m = len(p)
35     n = len(t)
36     d = n #long t
37     hp = 0
38     ht = 0
39     h = 1
40
41     for i in range(m-1):
42         h = (h*d) % q
43
44     # calculo hash value de p y t
45     for i in range(m):
46         hp = (d*hp + ord(p[i])) % q
47         ht = (d*ht + ord(t[i])) % q
48
49     #encuentro la subcadena en la cadena
50     for i in range(n-m+1):
51         if hp == ht:
52             for j in range(m):
53                 if t[i+j] != p[j]: #los hash coinciden pero los caracteres no
54                     break
55             j += 1
56             if j == m: #es porque no rompi el bucle anterior entonces encuentre la subcadena
57                 print(p,"found at ", str(i+1))
58                 break
59
60     if i < n-m: #si no, no podemos desplazarnos a la proxima subcadena porque no alcanza la long
61         ht = (d*(ht-ord(t[i])*h) + ord(t[i+m])) % q #a ht le resto el termino de mayor orden y le sumo el de menor orden
62         if ht < 0: #para que el hash no sea negativo
63             ht += q
64         else: return print("None")
65
66 """
```

## 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.

```
119 | #EJERCICIO 14                                KMP
120 |
121 | def KMP(t,p):
122 |     n = len(t)
123 |     m = len(p)
124 |     pi = computePrefixFunction(p)
125 |     q = 0
126 |     for i in range(0,n):
127 |         while q > 0 and p[q] != t[i]:
128 |             q = pi[q-1]
129 |             if p[q] == t[i]:
130 |                 q += 1
131 |             if q == m:
132 |                 print("Pattern occurs with shift", i-m+2)
133 |                 #q = pi[q] -----> en caso de que haya q buscar mas de una ocurrencia
134 |                 break
135 |             if q != m:
136 |                 return print("False")
137 |             else:
138 |                 return
139 |
140 | def computePrefixFunction(p):
141 |     m = len(p)
142 |     pi = [0]*m
143 |     pi[0] = 0
144 |     k = 0
145 |     for q in range(1,m):
146 |         while k > 0 and p[k] != p[q]:
147 |             k = pi[k]
148 |             if p[k] == p[q]:
149 |                 k += 1
150 |             pi[q] = k
151 |     return pi
152 |
```

## 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 = aabababaaa las ocurrencias de P aabababaaa y aabababaaa se solapan por lo que la mayor cantidad de ocurrencias no solapadas son 2, o sea aabababaaa.

`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.~~