

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

```
1 # c: Character
2 def existChar(String, c):
3     for i in String:
4         if i == c:
5             return True
6     return False
7
8 def isPalindrome(String):
9
10    string_sinEspacios = String.replace(" ", "")
11
12    mid_char = len(string_sinEspacios) // 2
13
14    for i in range(len(string_sinEspacios[:mid_char])):
15        if string_sinEspacios[i] != string_sinEspacios[len(string_sinEspacios) - i - 1]:
16            return False
17    return True
```

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.

```
8 def isPalindrome(String):
9
10     string_sinEspacios = String.replace(" ", "")
11
12     mid_char = len(string_sinEspacios) // 2
13
14     for i in range(len(string_sinEspacios[:mid_char])):
15         if string_sinEspacios[i] != string_sinEspacios[len(string_sinEspacios) - i - 1]:
16             return False
17     return True
18
```

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.

```
19 def mostRepeatedChar(String, c):
20
21     dict = {}
22
23     for i in String:
24         dict[i] += 1
25
26     return max(dict, key = dict.get)
27
```

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”

```
29 def reduceLen(String):
30
31     reduced_list = []
32     for char in String:
33
34         if char == (reduced_list and reduced_list[-1]):
35             reduced_list.pop()
36         else:
37             reduced_list.append(char)
38     return reduced_list
```

Ejercicio 8

Implementar una función que das 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 $s = \text{"aaaffmmmarillzzllhooo"}$ contiene **amarillo**, pero $s = \text{"aaaffmmmmarrilzzzhooo"}$ no (debido a que le falta una l). Si ordenamos la primera cadena como $s = \text{"aaaailllffzzzhrrmmmo"}$, 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.

```
40 def isContained(String, SubString):
41
42     long_SubString = len(SubString)
43
44     comprobation_array = [False] * long_SubString
45     i = 0
46
47     for char in String:
48         if char == SubString[i]:
49             comprobation_array[i] = True
50             i += 1
51         if i == long_SubString:
52             break
53     if comprobation_array.count(True) == long_SubString:
54         return True
55     else:
56         return False
57
```

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 “**cabccbcbacab**” 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 (\diamond) 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 $P = \text{"aabab"}$ y demostrar su funcionamiento en la cadena de texto $T = \text{"aaababaabaababaab"}$. **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 $O(n+m)$.

Ejercicio 12

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

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.

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.

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 aabaabaaa.

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.~~