Actividad 2.1 Operaciones con Strings

Victor Misael Escalante Alvarado, A01741176

Algoritmos para string-matching

KMP con LPS

Capturas del codigo

```python
# Algoritmo KMP con LPS
# Complejidad del peor caso: O(n + m), donde n es el tamaño del texto y m del patrón.

def compute_lps(pattern):
    """
    Calcula el array de Longest Prefix Suffix (LPS).
    """
    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1

    return lps
```

```python
def kmp_search(text, pattern):
    """
    Realiza la búsqueda del patrón en el texto usando KMP.
    """
    n, m = len(text), len(pattern)
    lps = compute_lps(pattern)
    i = j = 0  # i para texto, j para patrón
    matches = []

    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1

        if j == m:
            matches.append(i - j)  # Encuentra coincidencia
            j = lps[j - 1]
        elif i < n and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1

    return matches

# Ejemplo de uso
text = "ababcabcabababd"
pattern = "ababd"
print(kmp_search(text, pattern))  # Salida: [10]
```

Resultados

```
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ % "/Users/
ey/Sem 5/Algoritmos_/Scripts/Python/Act21.py"
[10]
[0, 0, 2, 0, 0, 0, 4, 0, 2, 0, 0, 4, 0, 5, 0, 2, 0, 0]
bab
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ %
```

Z-Algorithm con Z-array

Capturas del codigo

```python
# Z-Algorithm
# Complejidad del peor caso: O(n + m), donde n es el tamaño del texto y m del patrón.
def z_algorithm(s):
    """
    Calcula el Z-array del string.
    """
    n = len(s)
    z = [0] * n
    l, r, k = 0, 0, 0

    for i in range(1, n):
        if i > r:
            l, r = i, i
            while r < n and s[r] == s[r - l]:
                r += 1
            z[i] = r - l
            r -= 1
        else:
            k = i - l
            if z[k] < r - i + 1:
                z[i] = z[k]
            else:
                l = i
                while r < n and s[r] == s[r - l]:
                    r += 1
                z[i] = r - l
                r -= 1

    return z

# Ejemplo de uso
s = "ababcababababd"
pattern = "ababd"
z_input = pattern + "$" + s
z_array = z_algorithm(z_input)
print(z_array)  # Salida: Z-array para el string
```

Resultados

```
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ % "/Users/
ey/Sem 5/Algoritmos_/Scripts/Python/Act21.py"
[10]
[0, 0, 2, 0, 0, 0, 4, 0, 2, 0, 0, 4, 0, 5, 0, 2, 0, 0]
bab
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ %
```

Algoritmo Manacher para el palíndromo mas largo de una cadena.

Capturas del codigo

```python
# Algoritmo de Manacher
# Complejidad del peor caso: O(n), donde n es el tamaño del string.

def manacher(s):
    """
    Encuentra el palíndromo más largo en un string.
    """
    # Transformar el string para manejar palíndromos pares e impares.
    t = "#".join(f"^{s}$")
    n = len(t)
    p = [0] * n
    center = right = 0

    for i in range(1, n - 1):
        mirror = 2 * center - i
        if i < right:
            p[i] = min(right - i, p[mirror])

        # Expansión alrededor del centro
        while t[i + p[i] + 1] == t[i - p[i] - 1]:
            p[i] += 1

        # Actualizar el centro si es necesario
        if i + p[i] > right:
            center, right = i, i + p[i]

    # Obtener el palíndromo más largo
    max_len = max(p)
    center_index = p.index(max_len)
    start = (center_index - max_len) // 2
    return s[start:start + max_len]

# Ejemplo de uso
s_manacher = "babad"
print(manacher(s_manacher))  # Deberia salira: "bab"
```

Resultados

```
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ % "/Users/
ey/Sem 5/Algoritmos_/Scripts/Python/Act21.py"
[10]
[0, 0, 2, 0, 0, 0, 4, 0, 2, 0, 0, 4, 0, 5, 0, 2, 0, 0]
bab
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ %
```

Algoritmo para obtener el suffix array(fuerza bruta).

Capturas del codigo

```python
# Ejemplo de uso
s_manacher = "babad"
print(manacher(s_manacher))  # Deberia salira: "bab"

# Algoritmo para obtener el Suffix Array por fuerza bruta
# Complejidad del peor caso: O(n^2 log n), donde n es el tamaño del string.

def suffix_array(s):
    """
    Calcula el suffix array de un string.
    """
    n = len(s)
    suffixes = [(s[i:], i) for i in range(n)]
    suffixes.sort()  # Ordenar sufijos lexicográficamente
    return [suffix[1] for suffix in suffixes]

# Ejemplo de uso
s = "banana"
print(suffix_array(s))  # Salida: [5, 3, 1, 0, 4, 2]
```

Resultados

```
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ % "/U
ey/Sem 5/Algoritmos_/Scripts/Python/Act21.py"
[10]
[0, 0, 2, 0, 0, 0, 4, 0, 2, 0, 0, 4, 0, 5, 0, 2, 0, 0]
bab
[5, 3, 1, 0, 4, 2]
(.venv) misa_v@MacBook-Pro-de-Victor-7 Algoritmos_ %
```

¿Para que sirve el suffix array?

El suffix array es una estructura que ordena los sufijos de una cadena lexicográficamente, siendo útil en diversas aplicaciones como búsqueda eficiente de patrones al combinarse con el LCP, compresión de texto (base para algoritmos como Burrows-Wheeler Transform), análisis genómico y alineación de secuencias en bioinformática, así como para calcular rápidamente el número de substrings únicas en un string.

Enlace a codespaces: https://colab.research.google.com/drive/1eoo615XCocw7Xp2jHNAg61ZcVwHW45B5?usp=sharing