Taller 2
Complejidad Temporal

Este taller tiene como objetivo ayudarte a entender cómo medir y analizar la eficiencia de los algoritmos mediante la complejidad temporal, un concepto fundamental en programación. La complejidad temporal nos permite estimar cuánto tiempo tardará un algoritmo en ejecutarse en función del tamaño de la entrada, y nos ayuda a comparar diferentes soluciones y seleccionar la más adecuada para cada problema. En este taller, exploraremos diferentes tipos de complejidad temporal, aprenderemos a calcularla y analizarla, y practicaremos con algunos ejercicios prácticos. Al final del taller, tendrás una mejor comprensión de cómo optimizar tus algoritmos y escribir programas más eficientes.

Problema 1
Dado un array de n elementos, escribe un algoritmo para encontrar el par de elementos en el array que sumen el número más grande.
Solución (Código fuente)

```python
def maxSum(array):
    maxVal = 0
    maxVal2 = 0
    for element in array:
        if element > maxVal2:
            maxVal2 = element

        if maxVal2 > maxVal:
            temp = maxVal
            maxVal = maxVal2
            maxVal2 = temp

        print("valores: ", maxVal, maxVal2)
    return maxVal + maxVal2
```

Complejidad temporal (Explicación)
La complejidad temporal del algoritmo está determinada por el número de operaciones que realiza el algoritmo. En el caso del algoritmo maxSum, el algoritmo realiza 2n comparaciones. Por lo tanto, la complejidad temporal del algoritmo es O(n).

Formatted: Centered

Formatted: Left

Problema 2
Dado un array de n elementos, escribe un algoritmo para encontrar el subarray contiguo con la suma más grande.
Solución (Código fuente)

```python
def maxCGroup(array):
    maxsum = 0
    positions = []
    for i in range(0, len(array)-1):
        if (array[i] + array[i+1]) > maxsum:
            maxsum = array[i] + array[i+1]
            print("f", (array[i] + array[i+1]))
            positions = [array[i], array[i+1]]
    return positions
```

Complejidad temporal (Explicación)
La complejidad temporal del algoritmo está determinada por el número de operaciones que realiza el algoritmo. En el caso del algoritmo maxCGroup, el algoritmo realiza 2n operaciones. Por lo tanto, la complejidad temporal del algoritmo es O(n).

Formatted: Centered

Formatted: Font: Not Bold

Formatted: Left

Problema 3
Dado un array de n elementos, escribe un algoritmo para encontrar todos los pares de elementos en el array que sumen un número dado.
Solución (Código fuente)

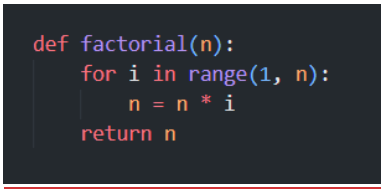
```python
def sumPares(array, n):
    positions = []
    for i in range(0, len(array)-1):
        for j in range(i, len(array)-1):
            if (array[i]+array[j] == n) and i != j:
                positions.append([i, j])
    return positions
```

Complejidad temporal (Explicación)
La complejidad temporal del algoritmo está determinada por el número de operaciones que realiza el algoritmo. En el caso del algoritmo sumPares, el algoritmo realiza n operaciones en cada iteración del ciclo for externo y n-1 operaciones en cada iteración del ciclo for interno. Por lo tanto, el algoritmo realiza un total de n^2 operaciones

Formatted: Centered

Formatted: Left

Formatted: Font: Not Bold, Spanish (Colombia)

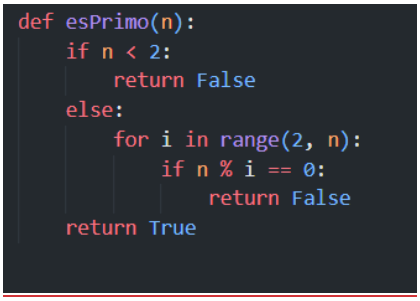
Problema 4
Dado un número entero n, escribe un algoritmo para calcular el factorial de n.
Solución (Código fuente)

Complejidad temporal (Explicación)
El algoritmo que proporcionaste para la función factorial es correcto. Tiene una complejidad temporal de O(n), ya que realiza un único ciclo for que itera de 1 a n - 1, y multiplica el valor actual de n por i en cada iteración.

```
def factorial(n):
    for i in range(1, n):
        n = n * i
    return n
```

Formatted: Centered

Formatted: Left

Formatted: Font: Not Bold

Problema 5
Dado un número entero n, escribe un algoritmo para verificar si n es un número primo.
Solución (Código fuente)

Complejidad temporal (Explicación)
La complejidad temporal de este algoritmo es O(n), ya que el bucle for se ejecutará al menos una vez, y el número de iteraciones del bucle será igual a n-2, que es proporcional al tamaño de la entrada n.

```
def esPrimo(n):
    if n < 2:
        return False
    else:
        for i in range(2, n):
            if n % i == 0:
                return False
    return True
```

Formatted: Centered

Formatted: Left

Formatted: Font: Not Bold, Spanish (Mexico)