Hoja de Trabajo HT-M1-1 - Solución

Sección 4 - EDA 2025-20

Objetivo:

- Practicar y afianzar el aprendizaje de los conceptos de:
  - Complejidad temporal.
  - Complejidad espacial.
  - Recursividad.

Bono:

- Esta hoja de trabajo es **bonificable**.
- Para obtener un bono usted **debe**:
  - Trabajar de forma **completamente individual y en silencio**.
  - No utilizar ningún tipo de ayuda:
    - No se permite el uso de dispositivos electrónicos.
  - Responder correctamente todas las preguntas.
  - Entregar la hoja de trabajo diligenciada a Manuela.
    - No se aceptarán entregas tardías o hechas por otros medios.
- Si alguno, su bono será aplicado a la **nota del parcial del módulo 1**.

Complejidad temporal y espacial

1. Considere la siguiente función:

```python
def suma_lista(lista):
    suma = 0
    for elemento in lista:
        suma += elemento

    return suma
```

1.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O?

Respuesta: 0(n)

Explicación: El ciclo `for` recorre todos los elementos de `lista` una sola vez. Si la longitud de la lista es n, el número de operaciones crece proporcionalmente a n.

Complejidad espacial de la función: **O(1).** Esto porque todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

2. Considere la siguiente función:

```
def dar_elemento_en_posicion(lista, pos):
    elemento = None
    if 0 <= pos < len(lista):
        elemento = lista[pos]

    return elemento
```

2.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O?

Respuesta: 0(1)

Explicación: Dos comparaciones: `0 <= pos` y `pos < len(lista)` → O(1). `len(lista)` → O(1). - Acceso indexado `lista[pos]` → O(1). Asignación a la variable `elemento` y `return` → O(1).

2.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Respuesta: 0(1)

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

3. Considere la siguiente función:

```
def busqueda_secuencial(lista, elemento):
    i = 0
    pos = -1
    encontrado = False

    while i < len(lista) and not encontrado:
        if lista[i] == elemento:
            pos = i
            encontrado = True
        else:
            i += 1

    return pos
```

3.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O?

Respuesta: 0(n)

Explicación: Todas las operaciones por fuera del ciclo son O(1). En cada iteración del `while` se hacen operaciones O(1). En el peor caso, se visita a todos los elementos de la lista → O(n).

3.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Respuesta: 0(1)

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

4. Considere la siguiente función:

```
def busqueda_matriz(matriz, filas, columnas, elemento):
    f = 0
    c = 0
    pos = (-1, -1)
    encontrado = False

    while f < filas and not encontrado:
        c = 0
        while c < columnas and not encontrado:
            if matriz[f][c] == elemento:
                pos = (f, c)
                encontrado = True
            else:
                c += 1
        f += 1

    return pos
```

4.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O, si se asume que la matriz es cuadrada?

Respuesta: $O(n^2)$

Explicación: Todas las operaciones por fuera de los ciclos son O(1). Peor caso: Se recorre toda la matriz → $n \cdot m$ visitas → $O(n \cdot m)$. Esto se puede aproximar en una matriz cuadrada a: $O(n^2)$

4.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Respuesta: 0(1)

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

5. Considere la siguiente función:

```
def contar_mayores_menores(lista, elemento):
    mayores = 0
    menores = 0

    for item in lista:
        if item > elemento:
            mayores += 1
        if item < elemento:
            menores += 1

    return mayores, menores
```

5.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O?

Respuesta: O(n)

Explicación: Todas las operaciones por fuera del ciclo son O(1). En cada iteración del `for` se hacen operaciones O(1). Se visita una vez a todos los elementos de la lista → O(n).

5.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Respuesta: 0(1)

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

6. Considere la siguiente función:

```
def contar_tripletas_suma_cero(lista):
    contador = 0

    for i in range(len(lista)):
        for j in range(i + 1, len(lista)):
            for k in range(j + 1, len(lista)):
                if lista[i] + lista[j] + lista[k] == 0:
                    contador += 1

    return contador
```

6.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O? Justifique su respuesta

Respuesta: O(n^3)

Justificación: Todas las operaciones por fuera de los ciclos son O(1). Hay tres ciclos anidados (`i`, `j`, k) → número de iteraciones ≈ O(n^3)

6.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O? Justifique su respuesta

Respuesta: 0(1)

Justificación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

7. Considere las dos siguientes funciones:

```python
def incognita_v1(n):
    suma = 0
    while n > 0:
        suma += n
        n = n // 2
    return suma

def incognita_v2(n):
    suma = 0
    while n > 0:
        for _ in range(n):
            suma += 1
        n = n // 2
    return suma
```

7.1 ¿Cuál de las dos funciones es más eficiente, considerando su complejidad temporal? Justifique su respuesta.

Respuesta: `incognita_v1`

Justificación:

- `incognita_v1`: Usa un ciclo mientras divide n a la mitad → ≈ $\lfloor \log_2 n \rfloor$ iteraciones, con operaciones O(1) por iteración → **O(log n)**.
- `incognita_v2`: En cada iteración recorre n, luego n/2, luego n/4, ... → suma geométrica n + n/2 + n/4 + … ≈ 2n → **O(n)**.

7.2 ¿Cuál es la complejidad espacial de ambas funciones en la notación Big-O? Justifique su respuesta.

Respuesta: 0(1)

Justificación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

8. Considere la siguiente función:

```python
def busquedas_binarias_en_lista(lista_ordenada, elementos_a_buscar):
    resultados = []

    for objetivo in elementos_a_buscar:
        i = 0
        f = len(lista_ordenada) - 1
        pos = -1

        while i <= f:
            m = (i + f) // 2
            if lista_ordenada[m] == objetivo:
                pos = m
                i = f + 1
            elif lista_ordenada[m] > objetivo:
                f = m - 1
            else:
                i = m + 1
        resultados.append(pos)

    return resultados
```

8.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O?

Respuesta: O(n log m)

Explicación: Todas las operaciones por fuera de los ciclos son O(1). Cada búsqueda binaria → O(log n), es repetida m veces (por el ciclo `for`).

8.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Respuesta: **0(n)**

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

Recursividad y Complejidades

A continuación, se presentan dos funciones recursivas solo a modo de referencia:

```python
def factorial(n):
    """
    Calcula el factorial de un número.

    :param n: El número positivo para calcular su factorial.
    :type n: int

    :returns: El factorial del número proporcionado.
    :rtype: int

    >>> factorial(0)  # Caso base
    1
    >>> factorial(1)  # Caso base
    1
    >>> factorial(2)  # Caso recursivo con 2
    2
    >>> factorial(3)  # Caso recursivo con 3
    6
    >>> factorial(4)  # Caso recursivo con 4
    24
    >>> factorial(5)  # Caso recursivo con 5
    120
    """
    return 1 if n <= 1 else n * factorial(n - 1)


def fibonacci(n):
    """
    Calcula el n-ésimo número de la secuencia de Fibonacci.

    :param n: La posición en la secuencia de Fibonacci (entero no negativo).
    :type n: int

    :returns: El n-ésimo número de Fibonacci.
    :rtype: int

    >>> fibonacci(0)  # Caso base
    0
    >>> fibonacci(1)  # Caso base
    1
    >>> fibonacci(2)  # Caso recursivo con 2
    1
    >>> fibonacci(12)  # Caso recursivo con 12
    144
    >>> fibonacci(24)  # Caso recursivo con 24
    46368
    """
    return n if n <= 1 else fibonacci(n - 1) + fibonacci(n - 2)
```

9. Ahora, **estudie** la función recursiva `sum_n()`. La función calcula la suma de todos los enteros desde 1 hasta un número positivo n recibido como parámetro, siguiendo este patrón:

$\sum^{in}_{i=1}$ i: 1

$\sum^{in}_{i=1}$ i: 1 + 2 = 3

$\sum^{in}_{i=1}$ i: 1 + 2 + 3 = 6

$\sum^{in}_{i=1}$ i: 1 + 2 + 3 + 4 = 10

$\sum^{in}_{i=1}$ i: 1 + 2 + 3 + 4 + 5 = 15

Implementación de la función:

```python
def sum_n(n: int) -> int:
    return 1 if n == 1 else n + sum_n(n - 1)
```

Complejidad temporal de la función: **O(n)**.

- La función se llama recursivamente con `n - 1` hasta llegar a 1.
- Número de llamadas recursivas: n.
- En cada llamada: una comparación (`n == 1`) y una suma → O(1).
- Total: `n × O(1)` → O(n).

Complejidad espacial de la función: **0(n)**

- Todas las variables ocupan un espacio constante.
- Cada llamada recursiva queda en el Call Stack de Python, hasta que se resuelve → profundidad de recursión = n.
- Total: O(n) por espacio ocupado en el Call Stack.

Aclaración: La anterior fue solo una implementación de ejemplo con fines de estudio. No es eficiente para valores grandes de n. Para calcular eficientemente la suma de un rango de enteros positivos, se usaría la siguiente fórmula: n(n + 1) / 2

```python
def sum_n(n: int) -> int:
    return n * (n + 1) // 2
```

Complejidad temporal de la función: **O(1)**.

- La función no usa ciclos ni llamadas recursivas.
- Solo realiza 3 operaciones aritméticas:
    - Multiplicación: `n * (n + 1)` → **O(1)**.
    - Suma: `(n + 1)` → **O(1)**.
    - División entera: `// 2` → **O(1)**.

Complejidad espacial de la función: **O(1)**.

- Todas las variables ocupan un espacio constante.
- No se crean estructuras adicionales cuyo tamaño dependa de n.
- No se usa el Call Stack de Python (sin recursión).

10. Implemente la función `contar_digitos()` recursivamente. La función debe contar la cantidad de dígitos en un número entero mayor a cero, recibido como parámetro.

- Ej:
  - `contar_digitos(7)` debe retornar `1`
  - `contar_digitos(12345)` debe retornar `5`

Requisitos:

- Use `n // 10` para eliminar el último dígito
- Repetir recursivamente
  - Hasta que el número tenga un solo dígito

Respuesta:

```
def contar_digitos(n: int) -> int:
    return 1 if n < 10 else 1 + contar_digitos(n // 10)
```

10.1 ¿Cuál es la complejidad temporal de su función en la notación Big-O? Justifique su respuesta

Respuesta: **O(log n)**

Justificación: Cada llamada recursiva divide n entre 10 → reduce el número de dígitos en 1. Número de llamadas ≈ cantidad de dígitos → $\log_{10}(n)$ → O(log n).

10.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O? Justifique su respuesta

Respuesta: **O(log n)**

Justificación: Cada llamada recursiva se mantiene en el Call Stack de Python hasta terminar. Profundidad de recursión = número de dígitos ≈ $\log_{10}(n)$ → O(log n).

Aclaración: La implementación recursiva solo se pide con fines de práctica. No es eficiente para valores grandes de n. Para calcular eficientemente se usaría: `int(math.log10(n)) + 1`. El logaritmo base 10 de un número, indica el orden de magnitud en potencias de 10, así:

```
def contar_digitos(n: int) -> int:
    return 1 if n == 0 else int(math.log10(n)) + 1
```

10.3 ¿Cuál es la complejidad temporal de la anterior función en la notación Big-O?

Tip: `math.log10(n)` e `int()` son operaciones constantes.

Respuesta: O(1)

Explicación: Llamar a `math.log10(n)` es una operación de tiempo constante. La suma, conversión a entero y demás operaciones también son O(1).

10.4 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Respuesta: O(1)

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

11. Implemente la función `sumar_digitos()` recursivamente. Esta función suma recursivamente los dígitos de un número entero no negativo recibido como parámetro.
    - Ej:
        - `sumar_digitos(5)` debe retornar 5
        - `sumar_digitos(123)` debe retornar 6

Respuesta:

```python
def sumar_digitos(n: int) -> int:
    return n if n < 10 else (n % 10) + sumar_digitos(n // 10)
```

11.1 ¿Cuál es la complejidad temporal de la función en la notación Big-O?

Posible respuesta (depende de la implementación): O(log n)

Explicación: Cada llamada recursiva divide n entre 10 → reduce el número de dígitos en 1. Número de llamadas ≈ cantidad de dígitos → $\log_{10}(n)$ → O(log n).

11.2 ¿Cuál es la complejidad espacial de la función en la notación Big-O?

Posible respuesta (depende de la implementación): O(log n)

Explicación: Cada llamada recursiva se mantiene en el Call Stack de Python hasta terminar. Profundidad de recursión = número de dígitos ≈ $\log_{10}(n)$ → O(log n).

Aclaración: Esta es solo una implementación de ejemplo con fines de práctica. No es eficiente para valores grandes de n.

11.3 Implemente una versión iterativa de la función `sumar_digitos`.

Posible respuesta:

```python
def suma_digitos_iter(n):
    suma = 0
    while n > 0:
        suma += n % 10
        n //= 10
    return suma
```

11.4 ¿Cuál es la complejidad temporal de la función que implementó en la notación Big-O?

Posible respuesta (depende de la implementación): O(log n)

Explicación: El ciclo se ejecuta una vez por cada dígito de n. Número de dígitos ≈ $\log_{10}(n)$ → O(log n).

11.5 ¿Cuál es la complejidad espacial de la función que implementó en la notación Big-O?

Posible respuesta (depende de la implementación): O(1)

Explicación: Todas las variables ocupan un espacio constante. No se crean estructuras adicionales cuyo tamaño dependa de n. No se usa el Call Stack de Python (sin recursión).

Recuerde: Debe entregar esta hoja de trabajo a Manuela, a más tardar al finalizar la sesión (12:20pm e idealmente antes). No se aceptarán entregas tardías o hechas por otros medios.