uc3m | Universidad Carlos III de Madrid

Tema 3 - Análisis de Algoritmos

Hoja de Problemas

1) Implementa una función, *sum_list*, que reciba una lista de enteros y devuelva la suma de sus elementos. No está permitido usar la función sum de las listas de Python.
2) Realiza el análisis empírico de este programa.
    a) Como el objetivo de este análisis es medir el tiempo de ejecución para distintos tamaños de listas, te recomendamos que implementes una función auxiliar, *random_list*, que reciba un número entero positivo, n, y devuelva una lista con n enteros creados de forma aleatoria.
    b) Modifica la función sum_list para que además de devolver el resultado, también devuelva el tiempo de ejecución (te recomendamos que sea en milisegundos).
    c) Obten los tiempos para tamaños de listas: 10, 10^2, 10^3,...,10^8
    d) Con esos valores construye una gráfica.
    e) ¿El tiempo de ejecución depende del tamaño de la lista?
3) Calcula la función temporal, T(n), y su orden superior de la función sum_list. Indica también el peor y mejor caso.
4) Escribe una función, pair_0(), que reciba una lista de Python, a, y devuelva el número de pares (i, j) tales que i != j y a[i] + a[j] = 0. Calcula su función temporal T(n) y su orden superior. Además, indica su peor y mejor caso.

5) Escribe una función, multiply, que dos matrices y devuelva la matriz producto de ambas. Calcula su función temporal T(n) y su orden superior. Además, indica su peor y mejor caso. Nota: una matriz en Python se puede representar como una lista de listas. Por ejemplo, m=[[1,2],[3,0],[4,-1]] es una matriz de dimensión 3x2 (3 filas, 2 columnas). Calcula su función temporal T(n) y su orden superior. Además, indica su peor y mejor caso.
6) A partir de las implementaciones de listas enlazadas que estudiamos en el tema 2 (lista enlazada simple sólo con head, lista enlazada simple con head y tail, lista doblemente enlazada), elabora una tabla que compare la complejidad asintótica (Big O) para cada uno de los métodos del TAD lista en las tres implementaciones. **No tienes que calcular las funciones temporales.** Es suficiente con que observes

el código y trates de razonar qué orden de complejidad tiene cada método. Razona también sobre el mejor y peor caso para cada método.

7)

Soluciones:

Problemas 1 y 2:
https://github.com/isegura/OCWEDA2022/blob/main/TEMA3/tema3.py
Ver tema3.xlsx

Problema 3

```
def sum_list(a: list) -> int:
    result = 0              #1
    for x in a:
        result += x         #2 * n, siendo n el número de elementos en a
    return result           #1
```

T(n) = 2n + 2
O(n) = n
Si la lista no es vacía, no hay mejor ni peor caso, porque siempre será necesario recorrer la lista completa para obtener la suma de sus elementos.
Si la lista está vacía, podemos considerarlo como mejor caso.

Problema 4

```
def pair_0(a: list) -> int:
    result = 0                          # 1
    for i in range(len(a)):             # n veces
        for j in range(len(a)):         # n veces,
            if i!=j and a[i] + a[j] == 0:     #1+1+1+1+1+1
                result += 1                   # 2

    return result                       #1
```

T(n) = 1 + n*n*8 + 1 = $8n^2+2$
O(n) = n^2
Si la lista no es vacía, no hay mejor ni peor caso, porque siempre será necesario recorrer la lista completa para buscar los pares de elementos que suman 0
Si la lista está vacía, podemos considerarlo como mejor caso.

Es posible plantear una segunda solución (algo más eficiente), sin embargo, veremos que ambas soluciones pertenecen al mismo orden superior n^2

```
def pair_0(a: list) -> int:
    result = 0                          # 1
    for i in range(len(a)-1):           # n-1 veces, n = len(a)
        for j in range(i+1, len(a)):    # n-(i+1) veces,
            if a[i] + a[j] == 0:        # 1 + 1 +1 + 1
                result += 1             # 2

    return result                       #1
```

n-(i+1) para i = 0, 1, ... n-1 (n-1 veces)

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1 = $\sum(i = 1,..., n - 1)\ i$ (aplicando fórmula de guas ((n-1)*n) / 2 = (n^2-n)/2

T(n) = 1 + (n^2-n)/2*6 + 1 = 3n^2-3n +2
O(n) = n^2

Si la lista no es vacía, no hay mejor ni peor caso, porque siempre será necesario recorrer la lista completa para buscar los pares de elementos que suman 0
Si la lista está vacía, podemos considerarlo como mejor caso.

Puedes ver la implementación de la función en:
https://github.com/isegura/OCWEDA2022/blob/main/TEMA3/tema3.py

Problema 5:

```
def multiply(matrix1, matrix2):

    if len(matrix1[0]) != len(matrix2):                                    # 4
        print('Error: cannot be multiplied!!!')
```

```
        return None

    n_rows = len(matrix1)                                                    # 2
    n_columns = len(matrix2[0])                                              # 3
    product_matrix = [[0 for _ in range(n_columns)] for _ in range(n_rows)]
                                                                  # n_columns * n_rows

    for i in range(n_rows):                                          # n_rows
        for j in range(n_columns):                                   # n_columns
            for k in range(n_rows):                                  # n_rows
                product_matrix[i][j] += matrix1[i][k] * matrix2[k][j]        # 6

    return product_matrix                                                    # 1
```

$T(n) = 4 + 2 + 3 + (n\_rows*n\_columns) + 6*n\_rows^2*n\_columns + 1$

definimos n = max(n_rows, n_columns)

$T(n) = 4 + 2 + 3 + (n\_rows*n\_columns) + 6*n\_rows^2*n\_columns + 1 <$
$10 + n^2 + 6n^3$

$O(n) = n^3$

El mejor caso es que las matrices no se puedan multiplicar (la primera matriz tenga un número de filas distintos al número de columnas de la segunda matriz).
El peor caso es cuando las matrices sí se pueden multiplicar.

Puedes ver la implementación de la función en:
https://github.com/isegura/OCWEDA2022/blob/main/TEMA3/tema3.py

Problema 6:

métodos	**SList (sólo head)**	**SList (head y tail)**	**DList (head y tail)**
constructor	O(1)	O(1)	O(1)
len (usando _size)	O(1)	O(1)	O(1)
len (sin _size)	O(n)	O(n)	O(n)
is_empty()	O(1)	O(1)	O(1)
__str__	O(n)	O(n)	O(n)
add_first	O(1)	O(1)	O(1)
remove_first	O(1)	O(1)	O(1)
add_last	O(n)	O(1)	O(1)

remove_last	O(n)	O(n)	O(1)
getAt	O(n)	O(n)	O(n)
index	O(n)	O(n)	O(n)
insertAt	O(n)	O(n)	O(n)
removeAt	O(n)	O(n)	O(n)

Análisis mejor y peor caso de DList

funciones	**Mejor Caso**	**Peor Caso**	**No tiene mejor/peor caso**
constructor			O(1)
len (usando _size)			O(1)
len (sin _size)	Lista vacía, O(1)	Lista no vacía, O(n)	
isEmpty()			O(1)
__str__	Lista vacía, O(1)	Lista no vacía, O(n)	
add_first			No hay ni mejor ni peor caso, O(1)
remove_first			No hay ni mejor ni peor caso, O(1)
add_last			No hay ni mejor ni peor caso, , O(1)
remove_last			No hay ni mejor ni peor caso, O(1)
getAt	Lista vacía o index incorrecto, O(1)	Lista no vacía e index=len(self)-1, si la búsqueda comienza en head. O(n)	
index	Lista vacía. Un segundo mejor caso, podría ser cuando el elemento está en el primer nodo. O(1)	Lista no vacía y no existe el elemento en la lista, porque tendrá que recorrer toda la lista, o el elemento está al final de la lista O(n)	
insertAt	Lista vacía o index no correcto.	Lista no vacía, index=len(self)-1, si	

	Además, si trato de insertar en los extremos (index=0 o index=len(self), complejidad también será O(1), porque estoy llamando a funciones con complejidad O(1) add_first y add_last().	la búsqueda comienza desde head, tenemos que llegar al penúltimo nodo O(n)	
removeAt	Lista vacía o index no correcto. Además, si trato de borrar los extremos (index=0 o index=len(self)-1), la complejidad también será O(1), porque estoy llamando a funciones con complejidad O(1) remove_first y remove_last().	Lista no vacía, index=len(self)-2, si la búsqueda comienza desde head, tenemos que llegar al penúltimo nodo O(n)	

Análisis mejor y peor caso de SList con head y tail

funciones	**Mejor Caso**	**Peor Caso**	**No tiene mejor/peor caso**
constructor			O(1)
len (usando _size)			O(1)
len (sin _size)	Lista vacía, O(n)	Lista no vacía, tenemos que recorrer toda la lista, O(n)	
is_empty()			O(1)
__str__	Lista vacía, O(1)	Lista no vacía, tenemos que recorrer toda la lista, O(n)	
add_first			No hay ni mejor ni peor caso. O(1)
remove_first			No hay ni mejor ni

			peor caso. O(1)
add_last			No hay ni mejor ni peor caso. O(1)
remove_last	Lista vacía, O(1)	Lista no vacía, O(n), es necesario recorrer toda la lista para llegar al penúltimo nodo	
getAt	Lista vacía o index incorrecto. El caso index=0, también se puede considerar como mejor caso. O(1)	Lista no vacía e index=len(self)-1, O(n)	
index	Lista vacía. Un segundo mejor caso, podría ser cuando el elemento está en el primer nodo. O(1)	Lista no vacía y no existe el elemento en la lista, o es el último elemento, porque tendrá que recorrer toda la lista. O(n)	
insertAt	Lista vacía o index no correcto. Además, si trato de insertar en los extremos (index=0 o index=len(self), complejidad también será O(1), porque estoy llamando a funciones con complejidad O(1) add_first y add_last().	Lista no vacía, index=len(self)-1, tenemos que ir hasta el antepenúltimo nodo. O(n)	
removeAt	Lista vacía o index no correcto. Además, si trato de borrar el primer nodo (i=0), la complejidad también será O(1), porque estoy llamando a la función remove_first, con complejidad O(1).	Lista no vacía, index=len(self)-1, hay que recorrer la lista para alcanzar el el penúltimo nodo. O(n)	

Análisis mejor y peor caso de SList únicamente con head

funciones	**Mejor Caso**	**Peor Caso**	**No tiene mejor/peor caso**
constructor			O(1) X
len (usando _size)			O(1) X
len (sin _size)	Lista vacía, O(n)	Lista no vacía, O(n)	
is_empty()			O(1)
__str__	Lista vacía, O(1)	Lista no vacía, O(n)	
add_first			No hay ni mejor ni peor caso. O(1)
remove_first			No hay ni mejor ni peor caso. O(1)
add_last	Lista vacía, O(1)	Lista no vacía, tenemos que recorrer toda la lista, O(n)	
remove_last	Lista vacía, O(1)	Lista no vacía, O(n), es necesario recorrer toda la lista para llegar al penúltimo nodo	
getAt	Lista vacía o index incorrecto. El caso index=0, también se puede considerar como mejor caso. O(1)	Lista no vacía e index=len(self)-1, O(n)	
index	Lista vacía. Un segundo mejor caso, podría ser cuando el elemento está en el primer nodo. O(1)	Lista no vacía y no existe el elemento en la lista o es el último, porque tendrá que recorrer toda la lista. O(n)	
insertAt	Lista vacía o index no correcto. Además, si trato de insertar al principio de la lista (index=0), la complejidad también será O(1), porque estoy llamando a la función add_first con complejidad	Lista no vacía, index=len(self), tenemos que ir hasta el penúltimo nodo nodo, O(n)	

	O(1).		
removeAt	Lista vacía o index no correcto. Además, si trato de borrar el primer elemento, la complejidad también será O(1), porque estoy llamando a la función remove_first con complejidad O(1).	Lista no vacía, index=len(self)-1, hay que recorrer la lista hasta el penúltimo nodo. O(n)	