Ejercicio 1

Demuestre que $6n^3 \neq \boldsymbol{O}(n^2)$.

Supongamos que la cantidad de operaciones elementales de un algoritmo está dada por $T(n) = 6n^3$
Entonces, lo que plantea la desigualdad anterior es que el orden de complejidad en el peor de los casos del algoritmo no puede ser $O(n^2)$, lo cuál es cierto, pues no existe función potencial con potencia 2 tal que multiplicada por una constante cualquiera C acote a la función T cuando n tienda a infinito. Por lo tanto podemos asegurar que el enunciado es verdadero

Ejercicio 2

¿Cómo sería un array de números (mínimo 10 elementos) para el mejor caso de la estrategia de ordenación Quicksort(n) ?

El mejor de los casos de quicksort es cuando el pivote seleccionado siempre termina al centro de la lista, de tal forma que la complejidad es O(nlog(n))
Además, el peor de los casos es cuando los elementos ya se encuentran ordenados o inversamente ordenados, pues en este caso no se divide en n/2 el problema, sinó que se quita solo un elemento de la lista haciendo que el orden de complejidad en el peor caso sea O(n^2)
Por lo tanto el mejor de los casos sería en una lista del siguiente tipo
A = [1, 6, 5, 2, 10, 7, 8, 9, 4, 3]
de esta forma, si el pivote queda en el centro de la lista obtenemos O(nlog(n))

De hecho, esta respuesta depende de cómo se escoja el pivote, si se hace aleatoriamente entonces tenemos que no importa la entrada, a mayor cantidad de elementos de la lista tenemos O(nlog(n))

Ejercicio 3

Cuál es el tiempo de ejecución de la estrategia **Quicksort(A), Insertion-Sort(A)** y **Mergesort(A)** cuando todos los elementos del array A tienen el mismo valor?

En este caso, Insertion sort resulta ser el más rápido, con una complejidad O(n)
MergeSort se mantiene constante O(nlog(n))

QuickSort O(nlog(n))

Ejercicio 4

La idea es ordenar la lista, seleccionar el elemento del medio y luego colocar la mitad de la lista de menores a la izquierda y la otra mitad a la derecha y hago lo mismo con los mayores

```python
"""
Implementar un algoritmo que ordene una lista de elementos
donde siempre el elemento del medio de la lista contiene
antes que él en la lista la mitad de los elementos menores que él.
Explique la estrategia de ordenación utilizada.
"""

# Lo primero que voy a hacer es encontrar el valor medio de la lista de entrada,
# de esta manera me aseguro que van a existir elementos mas grandes y mas chicos
# para llevar a cabo tal tarea, voy a utilizar un algoritmo de ordenamiento y luego
# retirar el emenento del medio de la lista
a = [4, 5, 7, 1, 2, 6, 10, 8, 9, 3]

sorted_list = sorted(a)   # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
middle_index = len(sorted_list) // 2 - 1  # 10 // 2 - 1 = 4
middle_value = sorted_list[middle_index]

greater = sorted_list[middle_index+1:]
smaller = sorted_list[:middle_index]
smaller_count = len(smaller)
greater_count = len(greater)

# El índice del valor medio en la nueva lista debe ser el mismo
half_smaller = smaller_count // 2
half_greater = greater_count // 2

result = smaller[:half_smaller] + greater[:half_greater] + [middle_value] + smaller[half_smaller:] + greater[half_greater:]
print(result)
```

Ejercicio 5

Explicación en comentarios

```python
"""
Implementar un algoritmo Contiene-Suma(A,n) que recibe una
lista de enteros A y un entero n y devuelve True si existen en
A un par de elementos que sumados den n. Analice el costo computacional.
"""

def contains_add(numbers, n):

    """
    La idea del algoritmo se basa en complementos.
    Por ejemplo:
        n = 21
        y tenemos la siguiente lista
            [6, 7, 2, 8, 9, 13, 1]
        se observa que 23 = 13 + 8. Por lo que el complemento con 13
        da 23 - 13 = 8.
        Entonces, la idea es que para cada elemento de la lista vamos a
        comprobar si su complemento se encuentra en la lista de complementos,
        en caso de que si, significa que si se encuentra la posible suma
    """

    complements = []

    for number in numbers:

        if number >= n:
            continue

        if number in complements:
            return True

        complements.append(n - number)

    return False
```

Ejercicio 6

Radix sort es un algoritmo de ordenamiento no recursivo. Mi implementación funciona solamente con números enteros.
Lo que hago es analizar el dígito de cada uno de los números, y en base a este valor, agrego al número en una lista de 10 elementos (place_list) en el índice dígito. Luego, itero en place_list para obtener una nueva lista numbers, con los mismos números, pero ordenados de otra forma. Este proceso se repite hasta que todos los dígitos sean 0, llegado este punto podemos asegurar que la lista numbers se encuentra ordenada

```python

def radix_sort(numbers):

    def get_digit_at(n, digit_index):
        # Given a number, returns the digit at the given position
        # digit_index starts in 0
        return (n // 10**digit_index) % 10

    # List in wich the numbers are placed. It's a list of lists
    place_list = [[] for _ in range(10)]

    current_digit = 0

    # With this flag, the loop will stop if all the digit for
    # all the numbers is 0 at the current digit_index
    non_zero_max_digit = True

    while non_zero_max_digit:

        non_zero_max_digit = False

        # 1) Places the numbers in the list
        for number in numbers:
            digit = get_digit_at(number, current_digit)
            non_zero_max_digit |= digit != 0
            place_list[digit].append(number)

        # Increases the digit index for next iteration
        current_digit += 1

        # 2) Then adds back the numbers to the numbers list, ready for the next step
        i = 0
        while i < len(numbers):

            for list_index in range(len(place_list)):
                for number in place_list[list_index]:
                    numbers[i] = number
                    i += 1

                # Clears the list for the next iteration
                place_list[list_index] = []

if __name__ == "__main__":
    l = [45, 3, 3, 5, 3,5 ,76, 7,4, 2, 4,5 , 6]
    radix_sort(l)
    print(l)
```

Ejercicio 7

A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: T(n) = a T(n/b) + f(n) y otros 3 con el método maestro simplificado: $T(n) = a\,T(n/b) + n^c$

1. T(n) = 2T(n/2)+n^4

   Resuelvo por método maestro completo

   a = 2, b = 2, f(n) = n^4

   n^logb(a) = n^log2(2) = n < n^4

   No puedo aplicar el caso 1, pues se observa que n es menor que f(n), tampoco puedo aplicar el caso 2, pues n y f(n) no son iguales. Lo que me lleva al caso 3

   Si ε = 3, Ω(n^(1 + 3)) = f(n), Ω(n^4) = f(n). Ahora debo comprobar que af(n/b) ≤ cf(n)

   2f(n/2) = 2(n/2)*4 = 2/2^4 * n^4 = (1/8)f(n). Donde c = ⅛

   Se cumplen los requisitos del tercer caso, por lo tanto tenemos que la complejidad se puede expresar como:

   T(n) = Θ(n^4)

2. T(n) = 2T(7n/10) + n

   a = 2, b = 10/7, f(n) = n

   logb(a) = log(10/7) (2) ≈ 1,9433. Luego

   Descarto el segundo caso, pues n^(1,9433) ≠ n. El tercer caso también lo descarto, pues n^(1,9433) > n. Entonces me encuentro frente al primer caso.

   Busco un ε > 0 tal que f(n) = O(n^(1,9433 - ε)). Luego ε ≈ 0,9433. Se cumplen las condiciones del primer caso del método maestro, por lo que puedo establecer que

   T(n) = Θ(n^(1,9433))

3. $T(n) = 16T(n/4) + n^2$

   a = 16, b = 4, f(n) = n^2

   log4(16) = 2

   Si aplico el método maestro, nos encontramos con el segundo caso, pues

   f(n) = Θ(log4(16)) = Θ(n^2)

   Entonces la complejidad:

   T(n) = Θ(n^2 ln(n))

4. $T(n) = 7T(n/3) + n^2$

   Ahora con el método simplificado

   a = 7, b = 3, c = 2

   log3(7) ≈ 1,7712 < 2

   Tercer caso del simplificado. T(n) = Θ(n^2)

5. $T(n) = 7T(n/2) + n^2$

   a = 7, b = 2, c = 2

   log2(7) ≈ 2.8073 > 2

   Primer caso del simplificado. T(n) = Θ(n^log2(7))

6. $T(n) = 2T(n/4) + \sqrt{n}$

   a = 2, b = 4, c = ½

   log4(2) = ½

   Segundo caso del simplificado. T(n) = Θ(n^½ ln(n))