3.

```
def misterio(n)
1.   if n <= 1 then
2.       return 1
3.   a =misterio(n/3)+3*misterio(n/4)
4.   b = √n
5.   i = 1
6.   while i < b * n/3 : do
7.       if i <= 5
8.           a = a + 2*misterio(n/4)+5*misterio(n/4)
9.       for j in range (1, b)
10.          k = n
11.          while k > 1
12.              a = a * 3
13.              k = k/4
14.      i = i + 2
15.  return a + b
```

a) La ecuación de recurrencia para el tiempo de ejecución de la función misterio(n) es:

T(n) = T(n/3) + T(3n/4) + O(n^(1/2)) + O(nlogn)

donde:

- T(n/3) y T(3n/4) corresponden al tiempo de ejecución de las dos llamadas recursivas que se hacen a la función misterio.

- O(n^(1/2)) corresponde al tiempo de ejecución del ciclo while i < b * n/3. La complejidad temporal de ese ciclo while es O(n^(1/2)), ya que la variable b es igual a la raíz cuadrada de n, por lo que el ciclo se ejecuta a lo sumo b * n/3 veces. Como b = √n, esto significa que el ciclo se ejecuta a lo sumo n^(1/2) veces. Por lo tanto, la complejidad temporal de ese ciclo es O(n^(1/2)).

- O(nlogn) corresponde al tiempo de ejecución de los dos ciclos anidados. El ciclo exterior se ejecuta b veces, donde b = sqrt(n). Dentro de este ciclo, el ciclo interior se ejecuta hasta que k <= 1, donde k comienza con el valor n y se divide por 4 en cada iteración. Entonces, el ciclo interior se ejecuta un total de log_4(n) veces.El número total de operaciones en ambos ciclos anidados es el producto del número de iteraciones del ciclo exterior y el número de operaciones en el ciclo interior. Por lo tanto, el número total de operaciones es:

b * log_4(n)
= sqrt(n) * log_4(n)
= log_2(n) * (sqrt(n) / log_2(4))
= O(n^(1/2) * logn)
= O(nlogn)

b) Para determinar el comportamiento asintótico de T(n), podemos utilizar el método maestro de la teoría de la resolución de ecuaciones de recurrencia.

La ecuación de recurrencia para T(n) es:

T(n) = T(n/3) + T(n/4) + O(n^(1/2)) + O(n log n)

Podemos ver que el término dominante en el lado derecho de la ecuación de recurrencia es T(n/3) + T(n/4). Por lo tanto, podemos utilizar el caso 2 del método maestro, que dice que si la ecuación de recurrencia tiene la forma:

T(n) = a T(n/b) + f(n)

y si f(n) es O(n^c) para alguna constante c, entonces:

Si log_b(a) < c, entonces T(n) es O(n^c).
Si log_b(a) = c, entonces T(n) es O(n^c log n).
Si log_b(a) > c, entonces T(n) es O(n^(log_b(a))).

En nuestro caso, a = 2, b = 3/4, y f(n) = O(n^(1/2) + n log n).

Para log_b(a), podemos calcular:

log_b(a) = log_3/4(2) = log(2) / log(3/4) ≈ 5.18

Por lo tanto, tenemos log_b(a) > c, donde c = 1/2, ya que n^(1/2) es menor que n^c para n lo suficientemente grande.

Entonces, según el caso 3 del método maestro, T(n) es O(n^(log_b(a))) = O(n^5.18).

Por lo tanto, podemos concluir que el comportamiento asintótico de T(n) es O(n^5.18).

5.

a) Pseudocódigo:

```
function cycle_sort(array):
    n = length(array)
    for i = 0 to n-2 do
        item = array[i]
        pos = i

        # Encuentra la posición en la que se debe colocar el elemento en su posición correcta
        for j = i+1 to n-1 do
            if array[j] < item then
                pos = pos + 1

        # Si el elemento ya está en su posición correcta, continúa con el siguiente elemento
        if pos == i then
            continue

        # Coloca el elemento en su posición correcta
        while item == array[pos] do
            pos = pos + 1
        swap(array[pos], item)

        # Rota los elementos en el ciclo
        while pos != i do
            pos = i
            for j = i+1 to n-1 do
                if array[j] < item then
                    pos = pos + 1
            while item == array[pos] do
                pos = pos + 1
            swap(array[pos], item)

    return array

```

b) Prueba de escritorio con ⟨3, 5, 10, 6, 4, 2, 6, 1⟩ usando impresiones en python:

```
--------Iteración 1 --------
item = 3
pos = 0
-----------Ciclo para encontrar la posición correcta
j = 1 | pos = 0
j = 2 | pos = 0
j = 3 | pos = 0
j = 4 | pos = 0
j = 5 | pos = 1
j = 6 | pos = 1
j = 7 | pos = 2
-----------Colocación del item en la posición correcta
array[2]= 3 | array = [3, 5, 3, 6, 4, 2, 6, 1]
item = 10
-----------Rotar los elementos en el ciclo
pos = 0
----------------Ciclo for anidado
j = 1 | pos = 1
j = 2 | pos = 2
j = 3 | pos = 3
j = 4 | pos = 4
j = 5 | pos = 5
j = 6 | pos = 6
j = 7 | pos = 7
----------------Ciclo while anidado
array[7]= 10 | array = [3, 5, 3, 6, 4, 2, 6, 10]
item = 1
------
pos = 0
----------------Ciclo for anidado
j = 1 | pos = 0
j = 2 | pos = 0
j = 3 | pos = 0
j = 4 | pos = 0
j = 5 | pos = 0
j = 6 | pos = 0
j = 7 | pos = 0
----------------Ciclo while anidado
array[0]= 1 | array = [1, 5, 3, 6, 4, 2, 6, 10]
item = 3
------



--------Iteración 2 --------
item = 5
pos = 1
```

```
----------Ciclo para encontrar la posición correcta
j = 2 | pos = 2
j = 3 | pos = 2
j = 4 | pos = 3
j = 5 | pos = 4
j = 6 | pos = 4
j = 7 | pos = 4
----------Colocación del item en la posición correcta
array[4]= 5 | array = [1, 5, 3, 6, 5, 2, 6, 10]
item = 4
----------Rotar los elementos en el ciclo
pos = 1
----------Ciclo for anidado
j = 2 | pos = 2
j = 3 | pos = 2
j = 4 | pos = 2
j = 5 | pos = 3
j = 6 | pos = 3
j = 7 | pos = 3
----------Ciclo while anidado
array[3]= 4 | array = [1, 5, 3, 4, 5, 2, 6, 10]
item = 6
------
pos = 1
----------Ciclo for anidado
j = 2 | pos = 2
j = 3 | pos = 3
j = 4 | pos = 4
j = 5 | pos = 5
j = 6 | pos = 5
j = 7 | pos = 5
----------Ciclo while anidado
array[5]= 6 | array = [1, 5, 3, 4, 5, 6, 6, 10]
item = 2
------
pos = 1
----------Ciclo for anidado
j = 2 | pos = 1
j = 3 | pos = 1
j = 4 | pos = 1
j = 5 | pos = 1
j = 6 | pos = 1
j = 7 | pos = 1
----------Ciclo while anidado
array[1]= 2 | array = [1, 2, 3, 4, 5, 6, 6, 10]
item = 5
------
```

```
--------Iteración 3 --------
item = 3
pos = 2
-----------Ciclo para encontrar la posición correcta
j = 3 | pos = 2
j = 4 | pos = 2
j = 5 | pos = 2
j = 6 | pos = 2
j = 7 | pos = 2


--------Iteración 4 --------
item = 4
pos = 3
-----------Ciclo para encontrar la posición correcta
j = 4 | pos = 3
j = 5 | pos = 3
j = 6 | pos = 3
j = 7 | pos = 3


--------Iteración 5 --------
item = 5
pos = 4
-----------Ciclo para encontrar la posición correcta
j = 5 | pos = 4
j = 6 | pos = 4
j = 7 | pos = 4


--------Iteración 6 --------
item = 6
pos = 5
-----------Ciclo para encontrar la posición correcta
j = 6 | pos = 5
j = 7 | pos = 5


--------Iteración 7 --------
item = 6
pos = 6
-----------Ciclo para encontrar la posición correcta
j = 7 | pos = 6


[1, 2, 3, 4, 5, 6, 6, 10]
```

c) Inicialización: Antes de la primera iteración del ciclo, el arreglo tiene un solo ciclo de longitud n, lo que satisface la propiedad de que todos los ciclos excepto el primero están ordenados, ya que solo hay un ciclo en este momento.

   Mantenimiento: Durante la i-ésima iteración del ciclo, el elemento i-esimo se coloca en su posición correcta, lo que implica que el ciclo formado por los elementos anteriores a i-1 es ordenado. Además, el algoritmo garantiza que los ciclos son de longitud 1 al terminar cada iteración, ya que cada elemento se coloca en su posición correcta y se forma un ciclo con él mismo.

   Terminación: Después de la última iteración del ciclo, todos los elementos están en su posición correcta y el arreglo está completamente ordenado.

d) La complejidad de tiempo del algoritmo Cycle Sort es O(n^2), donde n es la longitud del arreglo. Esto se debe a que en el peor caso, cada elemento del arreglo debe ser comparado con todos los demás elementos y posiblemente intercambiado con ellos, lo que requiere un número cuadrático de operaciones. Sin embargo, en el mejor caso (cuando el arreglo ya está ordenado), la complejidad es O(n).

e) La complejidad de espacio del algoritmo Cycle Sort es O(1), ya que no se utiliza memoria adicional para realizar el ordenamiento, sino que se modifican los elementos dentro del mismo arreglo.

f) Sí, el algoritmo Cycle Sort es in-place, ya que no utiliza una cantidad adicional de memoria para realizar las operaciones de ordenamiento, sino que reordena los elementos dentro del mismo arreglo de entrada.

   Además, el algoritmo Cycle Sort es estable, lo que significa que mantiene el orden relativo de los elementos con valores iguales. En otras palabras, si hay dos elementos con el mismo valor, el algoritmo los deja en el mismo orden relativo que tenían antes de la ordenación. Esto se debe a que, en el proceso de encontrar la posición correcta para un elemento, el algoritmo busca todas las posiciones que son menores que la posición actual, lo que garantiza que el elemento se coloque en la última posición disponible antes de cualquier otro elemento que tenga el mismo valor.

6.

e)

Cota Inferior (cota baja):

En cada iteración del bucle exterior del algoritmo, se coloca un elemento en su posición correcta y se elimina un ciclo de tamaño k, donde k es el número de

elementos que son menores que el elemento que se está colocando en su posición correcta. Dado que en el peor de los casos, todos los elementos son distintos entre sí, cada ciclo eliminado debe tener al menos un elemento, lo que implica que el número de ciclos es al menos n. Como el número de comparaciones realizadas para eliminar un ciclo de tamaño k es k, el número total de comparaciones realizadas por el algoritmo en el mejor caso es al menos:

(n-1) + (n-2) + ... + 2 + 1 = (n-1)*n/2

Cota Superior (cota alta):

El peor caso se produce cuando el arreglo está ordenado en orden inverso, es decir, en el primer ciclo se deben realizar n-1 comparaciones, en el segundo ciclo n-2 comparaciones, y así sucesivamente. En total, el número de comparaciones realizadas por el algoritmo en el peor caso es:

(n-1) + (n-2) + ... + 2 + 1 = (n-1)*n/2

Además, en cada ciclo el algoritmo realiza un número constante de asignaciones y otras operaciones, lo que contribuye con O(n) operaciones en el peor caso. Por lo tanto, la cota superior (cota alta) del algoritmo de ordenamiento de ciclo es O(n^2).

Dado que la cota inferior y la cota superior coinciden, podemos decir que la complejidad temporal asintótica del algoritmo de ordenamiento de ciclo es Θ(n^2).