6.

a) Implementación de algoritmo Merge Sort en Python:

```
# Libraries

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from time import *
from datetime import date
```

```
def merge(left,right):

    temporal_sort = []


    while left and right:  # While the len of left and right > 0

        # Sort and combine
        temporal_sort.append((left if left[0] <= right[0] else right).pop(0))


    # Add the last values to have a complete sort array
    if 0 <= len(left):
        temporal_sort += (left)
    if 0 <= len(right):
        temporal_sort += (right)

    return temporal_sort


def merge_sort(unsorted_array):

    len_list = len(unsorted_array)

    if len_list == 1 or len_list == 0:
        return unsorted_array

    # Calculate the central point of the array

    mid = (len_list)// 2

    # Left Sort
    left_sort = merge_sort(unsorted_array[:mid])
    # Right Sort
    right_sort = merge_sort(unsorted_array[mid:])

    # Return the sorted array

    return merge(left_sort,right_sort)
```

b) Considerar diferentes valores de n, partiendo de 1 e ir incrementándose. Para cada valor de n hacer 100 experimentos y reportar la mediana de los tiempos de ejecución de estos (para dicho n). Se deben tomar valores de n hasta que el tiempo de ejecución total dure más de 5 minutos Específicamente, cada experimento consiste en los siguientes pasos:

a) Generar aleatoriamente un arreglo de enteros de valores de 1 a 5000

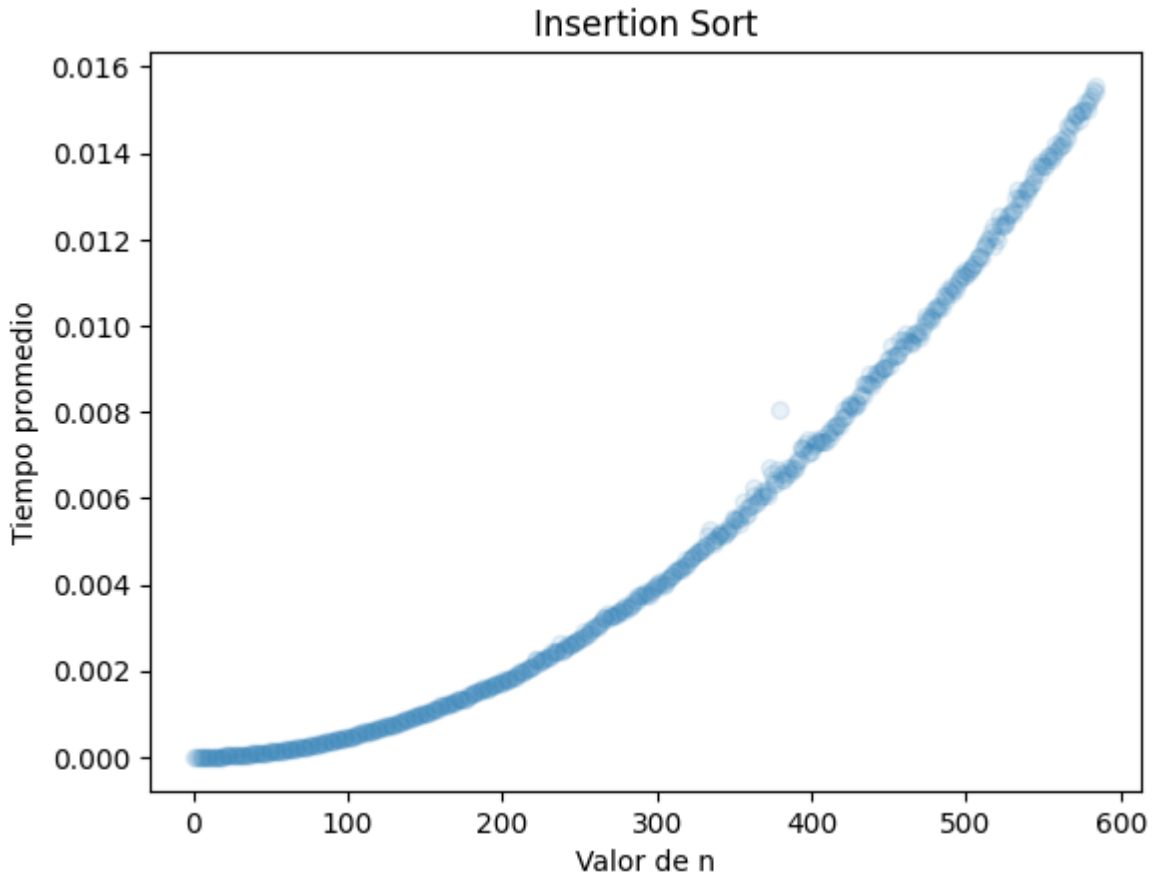
b) Calcular el tiempo de ejecución tomado por insertion-sort para ordenar ese arreglo en específico.

```
# Insertion Sort Algorithm

def insertion_sort(unsorted_array):

    for i in range(1,len(unsorted_array)):

        key = unsorted_array[i]
        j = i - 1

        while 0 <= j and unsorted_array[j] > key:
            unsorted_array[j+1] = unsorted_array[j]
            j -= 1
        unsorted_array[j+1] = key
    return unsorted_array
```

```
start_time = time()
end_time = start_time + (60*5)
n = 1
average_times = []

while time() < end_time:

    time_temp = []

    # A) Generar aleatoriamente un arreglo de enteros de valores de 1 a 5000

    for i in range(100):

        unsorted_array = np.random.randint(1,5000,n)

        start_time_sort = time()

        # Sort the unsorted array
        sorted_array = insertion_sort(unsorted_array)

        # Calculate the time for the Insertion Sort Algorithm
        time_execution = time() - start_time_sort

        time_temp.append(time_execution)

    average_times.append(np.average(np.array(time_temp)))

    n += 1
```

```
# Create a DataFrame to graph the different times
df_insertionSort = pd.DataFrame({"Size n":list(range(1,n)),"Average Time":average_times})

# Graph

plt.scatter(df_insertionSort['Size n'],df_insertionSort["Average Time"],marker = "o",label = "second",alpha= 0.1)
plt.xlabel("Valor de n")
plt.ylabel("Tiempo promedio")
plt.title("Insertion Sort")
plt.show();
```

c)



e) Cotas del algoritmo Cycle Sort:

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

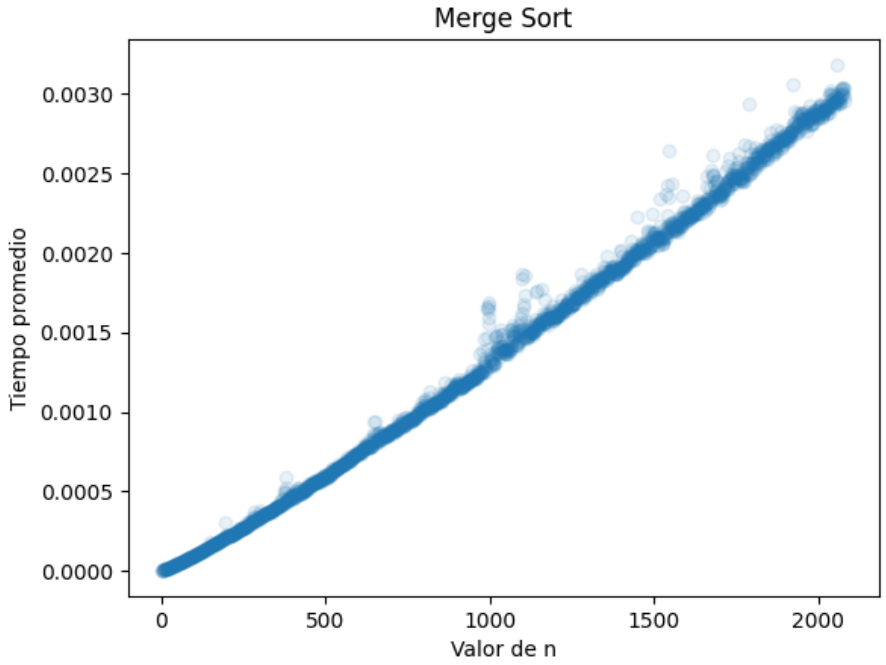
Además, en cada ciclo el algoritmo realiza un número constante de asignaciones y otras operaciones, lo que contribuye con O(n) operaciones en el peor caso. Por lo tanto, la cota superior (cota alta) del algoritmo de ordenamiento de ciclo es O(n^2).

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).

g) En una misma gráfica mostrar el tiempo experimental de los algoritmos merge-sort y cycle-sort para los diferentes valores de n trabajados.

```
start_time = time()
end_time = start_time + (60*5)
n = 1
average_times = []

while time() < end_time:

    time_temp = []

    # A) Generar aleatoriamente un arreglo de enteros de valores de 1 a 5000

    for i in range(100):

        unsorted_array = np.random.randint(1,5000,n)

        start_time_sort = time()

        # Sort the unsorted array
        sorted_array = merge_sort(list(unsorted_array))

        # Calculate the time for the Insertion Sort Algorithm
        time_execution = time() - start_time_sort

        time_temp.append(time_execution)

    average_times.append(np.average(np.array(time_temp)))

    n += 1
```

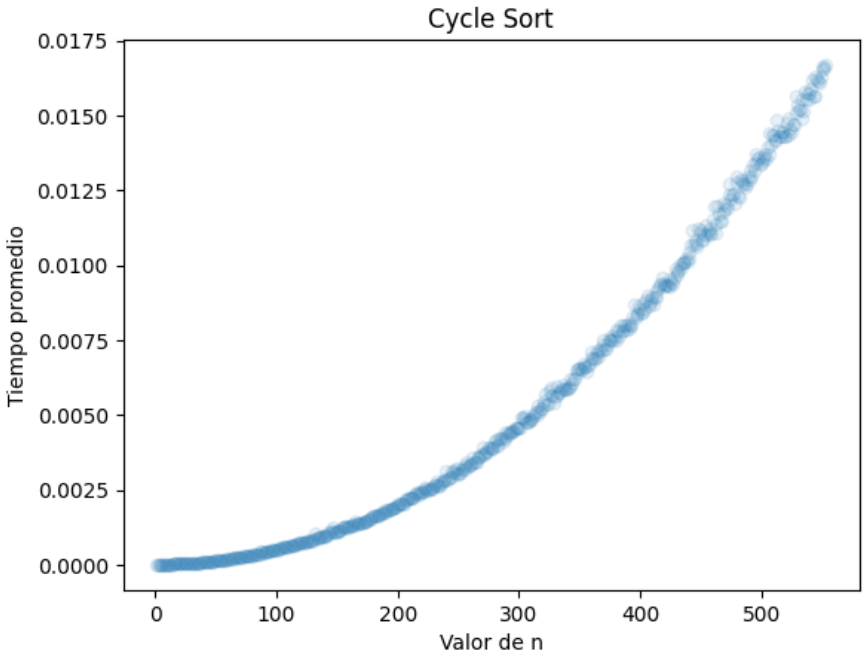
```
# Create a DataFrame to graph the different times
df_merge = pd.DataFrame({"Size n":list(range(1,n)),"Average Time":average_times})

# Graph

plt.scatter(df_merge['Size n'],df_merge["Average Time"],marker = "o",label = "second",alpha= 0.1)
plt.xlabel("Valor de n")
plt.ylabel("Tiempo promedio")
plt.title("Merge Sort")
plt.show();
```




```python
def cycle_sort(array):
    n = len(array)
    for i in range(0, n-1):
        item = array[i]
        pos = i

        # Find the position to put the item in its correct position
        for j in range(i+1, n):
            if array[j] < item:
                pos += 1

        # If the item is already in its correct position, continue to the next item
        if pos == i:
            continue

        # Place the item in its correct position
        while item == array[pos]:
            pos += 1

        array[pos], item = item, array[pos]

        # Rotate the elements in the cycle

        while pos != i:
            pos = i
            for j in range(i+1, n):
                if array[j] < item:
                    pos += 1
            while item == array[pos]:
                pos += 1
            array[pos], item = item, array[pos]
    return array
```

```
start_time = time()
end_time = start_time + (60*5)
n = 1
average_times = []

while time() < end_time:

    time_temp = []

    # A) Generar aleatoriamente un arreglo de enteros de valores de 1 a 5000

    for i in range(100):

        unsorted_array = np.random.randint(1,5000,n)

        start_time_sort = time()

        # Sort the unsorted array
        sorted_array = cycle_sort(list(unsorted_array))

        # Calculate the time for the Insertion Sort Algorithm
        time_execution = time() - start_time_sort

        time_temp.append(time_execution)

    average_times.append(np.average(np.array(time_temp)))

    n += 1
```

```
# Create a DataFrame to graph the different times
df_cycle = pd.DataFrame({"Size n":list(range(1,n)),"Average Time":average_times})

# Graph

plt.scatter(df_cycle['Size n'],df_cycle["Average Time"],marker = "o",label = "second",alpha= 0.1)
plt.xlabel("Valor de n")
plt.ylabel("Tiempo promedio")
plt.title("Cycle Sort")
plt.show();
```



```
# Create a DataFrame to graph the different times
df_cycle = pd.DataFrame({"Size n":list(range(1,n)),"Average Time":average_times})

# Graph

plt.scatter(df_cycle['Size n'],df_cycle["Average Time"],marker = "o",label = "second",alpha= 0.1)
plt.scatter(df_merge['Size n'],df_merge["Average Time"],marker = "o",label = "second",alpha= 0.1)
plt.xlabel("Valor de n")
plt.ylabel("Tiempo promedio")
plt.title("Cycle Sort")
plt.show();
```

GRÁFICA: