INSTITUTO POLITÉCNICO NACIONAL

Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas.

Materia:
Análisis y Diseño de Algoritmos

Práctica 02: Implementación y Evaluación del Algoritmo Quicksort.

Docente:
M. en C. Erika Sánchez-Femat

Nombre del alumno:
Dalia Naomi García Macías

Fecha de entrega:
23 de Octubre de 2023

Índice

1. Introducción

Desde que existe la ciencia de la computación, uno de los mayores problemas con los que los ingenieros se encontraban en su día a día, era el de ordenar listas de elementos. Por su causa, diversos algoritmos de ordenación fueron desarrollados a lo largo de los años y siempre existió un intenso debate entre los desarrolladores sobre cual de todos los algoritmos de ordenación era el más rápido.

El debate finalizó abruptamente en 1960 cuando Sir Charles Antony Richard Hoare, nativo de Sri Lanka y ganador del premio Turing en 1980, desarrolló el algoritmo de ordenación QuickSort casi por casualidad mientras ideaba la forma de facilitar la búsqueda de palabras en el diccionario.

El algoritmo QuickSort, en esapñol método de ordenamiento rápido, será el que se analizará durante la práctica, basa en la técnica de "divide y vencerás" por la que en cada recursión, el problema se divide en subproblemas de menor tamaño y se resuelven por separado (aplicando la misma técnica) para ser unidos de nuevo una vez resueltos.

2. Desarrollo de la Práctica

2.1. Implementación del algoritmo QuickSort

Como bien especificamos en el documento teórico de esta práctica, el algoritmo QuickSort es un código que nos ayuda a ordenar elementos de un arreglo. Esto lo hace con la técnica divide y vencerás. Consiste en ir dividiendo el arreglo original tomando un número que llamaremos pivote, número que tomará como referencia para dividir el arreglo original. Luego de tener el pivote definido, comparar este número con todos los demás elementos, de forma que los números menores a este pivote los colocará en un arreglo que pondrá del lado izquierdo del pivote. Por el contrario, los números mayores al pivote los pondrá en otro arreglo al lado derecho del pivote, y el pivote lo dejaremos solo en un arreglo. Después de haber hecho esto, repetiremos este paso con cada uno de los arreglos que se vayan generando, hasta que quede cada numero separado en un arreglo para, por último, juntamr todos esos números que ya estarán ordenados.

El algoritmo lo decidí dividir en funciones, una función diferente según cada especificación de la práctica. Primero, una función que me diera el tamaño de manera aleatoria de los arreglos; luego, creé una función que generara números aleatorios y los metiera dentro de un arreglo, esto según el tamaño que recibía de la primera función. Y, finalmente, una tercera función que crea una lista con todos los arreglos generados que salen de la función anterior. Finalmente, metí uno por uno los arreglos a la función que contendrá como tal el código principal de ordenamiento (QuickSort).

Listing 1: Funcion que define los tamanos del arreglo

```
#Funcion que genera el tama o de el arreglo
def Tam():
    #Genera el tama o aleatoriamente
    t=random.randrange(1,10)
    return t
```

Listing 2: Funcion que genera cada arreglo segun el tamaño del arreglo

```
#Funcion que genera el arreglo
def GenA(n):
```

```
    #Llena el arreglo aleatoriamente
    A=[random.randrange(1,20) for i in range(n)]
    return A
```

Listing 3: Funcion que guarda los 100 arreglos

```
#Funcion que genera una lista donde se guardan los 100 arreglos
def Arreglos():
    #Genera la lista vacia
    a=[]
    #Hace que nos genere tantos arreglos como queramos
    for i in range(100):
        #Va agregando a la lista cada arreglo generado
        a.append(GenA(Tam()))
    return a
```

Aparte de las funciones anteriores, sabemos que dentro del algoritmo QuickSort, una parte importante es el pivote, el cual se calculó sacando la media del arreglo, así que para esto también creé una función aparte, la cual luego se manda llamar a la función principal QuickSort.

Listing 4: Funcion que genera la media de cada arreglo

```
def Media(x):
    #Usa sum, nos ayuda a sumar todos los elementos de un arreglo
    #La doble linea redondea la division a un numero entero
    #Len calcula la longitud del arreglo
    m=sum(x)//len(x)
    return m
```

Ahora, dado que nuestro algoritmo debía ser recursivo, como caso base establecí que sería cuando el tamaño de los arreglos sea 1, y como caso general, hice un for. Este for es el que separa los números en los arreglos correspondientes al compararlos con el número que se toma como pivote, y se regresan a la función los nuevos arreglos que creamos llamados 'izq' y 'der', esto aparte de 'p', el arreglo que solo contiene el número que tomamos como pivote.

Listing 5: Funcion del QuickSort

```
def QuickSort(x):
    #Se define el caso base, que dice que cuando un arreglo solo tenga
        un elemento pues ya estara ordenado
    if len(x)<=1:
        return x
    #Se define el pivote, que ya se calculo anteriormente en la
       funcion media
    m=Media(x)
    p=m
    #Se definen los arreglos donde clasificaremos los elementos luego
       de las comparaciones
    izq=[]
    der=[]
    igual=[]
    #Cilco que asigna cada valor del arreglo original a los arreglos
       resultantes de la divison
```

```
    for e in x:
        if e<p:
            izq.append(e)
        elif e>p:
            der.append(e)
        else:
            #Se crea tambien un arreglo igual para el caso en que haya
                numeros repetidos o iguales al pivote
            igual.append(e)
    #Se definen los dos arreglos en una variable al momento de
       volverlas a llamar
    izq_ordenado=QuickSort(izq)
    der_ordenado=QuickSort(der)
    #Se regresan los arreglos resultantes de las divisiones del
       arreglo original y se concatenan
    return izq_ordenado + igual + der_ordenado
```

Listing 6: Implementacion de las funciones en el main

```
print("\n\t\t\tQuickSort")
n=Tam()
#Lista que contiene cada arreglo
x=Arreglos()

#Imprime cada elemento de la lista
print("\n-> Arreglos generados aleatoriamente")
Tama os=[]
for i in x:
    print(f"   El arreglo {x.index(i)} es: {i}")
    Tama os.append(len(i))

#Manda cada arreglo a la funcion QuickSort para que ordene de uno por
   uno
print("\n-> Arreglos ordenados")
ArregloOrdenado=[]
for i in x:
    m=Media(i)
    #Cada arreglo ordenado lo guarda en una variable
    ordenado=QuickSort(i)
    ArregloOrdenado.append(ordenado)
    #Imprime cada arreglo uno por uno ordenado
    print(f"   El arreglo {x.index(i)} ordenado es: {ordenado}")
```

2.2. Generación de casos de prueba

Para esta sección o este punto, fue entonces que creé las funciones, de modo que en la función 'lista', que guarda los arreglos generados, el límite fuera cien y generara así 100 arreglos con tamaños y números aleatorios.

A continuación mostraré el código y un ejemplo de su funcionamiento:

Listing 7: Funcion que guarda los 100 arreglos generados aleatoriamente

```
#Funcion que genera una lista donde se guardan los 100 arreglos
def Arreglos():
    #Genera la lista vacia
    a=[]
    #Hace que nos genere tantos arreglos como queramos
    for i in range(100):
        #Va agregando a la lista cada arreglo generado
        a.append(GenA(Tam()))
    return a
```

2.3. Medición del tiempo

Para el caso de la medición del tiempo, usé una función que calculaba el tiempo de ejecución cada vez que se ejecutaba la función QuickSort, ya que lo que nos importaba era cuánto se tardaba en ordenar cada arreglo. Lo que mandaba la función en cada caso era el tiempo que se calculaba por arreglo, y al final, para no perder el tiempo de cada arreglo, lo almacenaba en una lista llamada 'Tiempos', dado que todos estos tiempos se usarían después.

Listing 8: Funcion que calcula el tiempo de ejecucuion

```
#Funcion que genera una lista donde se guardan los 100 arreglos
#Funcion para calcular el tiempo de ejecucion de una funcion
def TiempoEjecucion(func, *args):
    #Tomamos el tiempo de ejecucion incial
    inicio=time.time()
    #Llama a la funcion de la que queremos el tiempo de ejecucion
    func(*args)
    #Tomamos el tiempo de ejecucion final
    fin=time.time()
    #Calculamos el tiempo de ejecucion real
    TiempoReal=fin-inicio
    #Regresamos como resultado de la funcion el tiempo de ejecucion
    return TiempoReal
```

Listing 9: Calculo del tiempo de ejecucuion en el main

```
print("\n-> Tiempos de ejecucion")
Tiempos=[]
Resultados=[]
for i in x:
    #Cada tiempo de ejecucion lo guardaremos en la variable tiempo
    Tiempo=TiempoEjecucion(QuickSort,i)
    #Imprime el tiempo de ejecucion de cada arreglo generado
    print(f"   Tiempo tardado en ordenar el arreglo {x.index(i)}: {
        Tiempo}")
    #Se guarda cada tiempo de ejecucuion en un arreglo
    Tiempos.append(Tiempo)
    Resultados.append((Tiempo, i, QuickSort(i)))
```

2.4. Visualización de datos

Para la visualización de datos, se usó la biblioteca Matplotlib de Python, donde se mandaron como datos la lista que creé con los tiempos de ejecución, además de poner nombres a los ejes y al gráfico.

Listing 10: Calculo del tiempo de ejecucuion en el main

```
plt.xlabel("Tama o del arreglo (n)")
plt.ylabel("Tiempo de ejecuci n (segundos)")
plt.title("Gr fico del algoritmo QuickSort")

# Usamos n como el tama o del arreglo en el eje X y Tiempos en el eje
    Y
plt.plot(Tiempos)
plt.show()
```

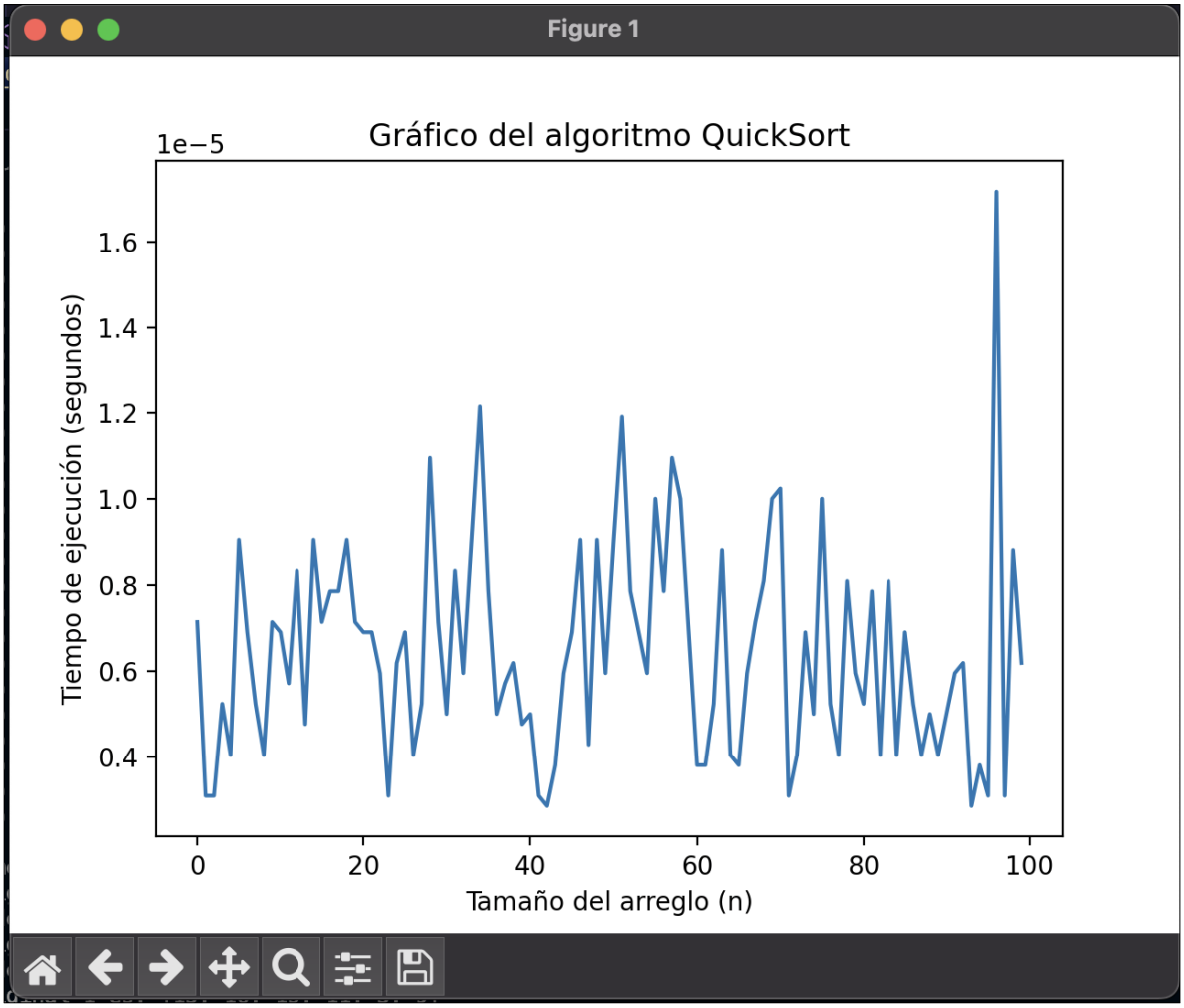

Figura 1: Grafico de los casos de prueba

Además de esto, al final se muestran los arreglos desordenados y ordenados de los 3 casos de prueba que tuvieron menor tiempo de ejecución y los 3 casos que tardaron más tiempo. Esto se logro

creando una lista con todos los tiempos de ejecucuion y sus posiciones, para luego ordenarlas de forma ascendente, de esta forma imprimir los tres primeros arreglo y los tres ultimos.

Listing 11: Calculo del tiempo de ejecucuion en el main

```
Resultados = sorted(Resultados, key=lambda x: x[0])

print("\n-> Arreglos con menor tiempo de ejecuci n")
for i, (tiempo, arreglo_original, arreglo_ordenado) in enumerate(
    Resultados[:3], 1):
    print(f"   El arreglo original {x.index(arreglo_original)} es: {
        arreglo_original}")
    print(f"   El arreglo ordenado {x.index(arreglo_original)} es: {
        arreglo_ordenado}")

print("\n-> Arreglos con mayor tiempo de ejecuci n")
for i, (tiempo, arreglo_original, arreglo_ordenado) in enumerate(
    Resultados[-3:], 1):
    print(f"   El arreglo original {x.index(arreglo_original)} es: {
        arreglo_original}")
    print(f"   El arreglo ordenado {x.index(arreglo_original)} es: {
        arreglo_ordenado}")
```

2.5. Módificacion del pivote

Para esta parte del código, lo único que hice fue dejar de usar la función que utilizaba aparte para calcular la media del arreglo y puse que usara como pivote el primer elemento del arreglo, por lo tanto solo se modifico la funcion de QuickSort, quedando asi:

Listing 12: Función QuickSort con la modificación del pivote

```
def QuickSort(x):
    #Se define el caso base, que dice que cuando un arreglo solo tenga
        un elemento pues ya estara ordenado
    if len(x)<=1:
        return x
    #Se define el pivote como el primer numero del arreglo
    p=x[0]
    #Se definen los arreglos donde clasificaremos los elementos luego
       de las comparaciones
    izq=[]
    der=[]
    igual=[]
    #Cilco que asigna cada valor del arreglo original a los arreglos
       resultantes de la divison
    for e in x:
        if e<p:
            izq.append(e)
        elif e>p:
            der.append(e)
        else:
```

```
            #Se crea tambien un arreglo igual para el caso en que haya
                numeros repetidos
            #O iguales al pivote
            igual.append(e)
    #Se definen los dos arreglos en una variable al momento de
       volverlas a llamar
    izq_ordenado=QuickSort(izq)
    der_ordenado=QuickSort(der)
    #Se regresan los arreglos resultantes de las divisiones del
       arreglo original
    return izq_ordenado + igual + der_ordenado
```

Para calcular una muestra de que los arreglo igual se siguen ordenando, hice un aprueba con 5 arreglo para no volver a poner capturas de 100, pero igual funciona para 100.

```
                        QuickSort

-> Arreglos generados aleatoriamente
   El arreglo 0 es: [16, 16, 10, 18, 8, 15, 17, 14, 12, 3]
   El arreglo 1 es: [12, 13, 5, 14, 7, 19, 18, 7, 13, 8]
   El arreglo 2 es: [4, 2, 12, 16, 3, 6, 19, 19, 15, 1, 15, 13]
   El arreglo 3 es: [19, 12, 19, 1, 9, 11, 12, 18, 2, 10, 17, 6, 11, 10]
   El arreglo 4 es: [19, 8, 17, 6, 15, 16, 4, 2, 7, 19, 2]

-> Arreglos ordenados
   El arreglo 0 ordenado es: [3, 8, 10, 12, 14, 15, 16, 16, 17, 18]
   El arreglo 1 ordenado es: [5, 7, 7, 8, 12, 13, 13, 14, 18, 19]
   El arreglo 2 ordenado es: [1, 2, 3, 4, 6, 12, 13, 15, 15, 16, 19, 19]
   El arreglo 3 ordenado es: [1, 2, 6, 9, 10, 10, 11, 11, 12, 12, 17, 18, 19, 19]
   El arreglo 4 ordenado es: [2, 2, 4, 6, 7, 8, 15, 16, 17, 19, 19]

-> Tiempos de ejecucion
   Tiempo tardado en ordenar el arreglo 0: 6.9141387939453125e-06
   Tiempo tardado en ordenar el arreglo 1: 6.9141387939453125e-06
   Tiempo tardado en ordenar el arreglo 2: 6.9141387939453125e-06
   Tiempo tardado en ordenar el arreglo 3: 1.0013580322265625e-05
   Tiempo tardado en ordenar el arreglo 4: 6.9141387939453125e-06

-> Arreglos con menor tiempo de ejecución
   El arreglo original 0 es: [16, 16, 10, 18, 8, 15, 17, 14, 12, 3]
   El arreglo ordenado 0 es: [3, 8, 10, 12, 14, 15, 16, 16, 17, 18]
   El arreglo original 1 es: [12, 13, 5, 14, 7, 19, 18, 7, 13, 8]
   El arreglo ordenado 1 es: [5, 7, 7, 8, 12, 13, 13, 14, 18, 19]
   El arreglo original 2 es: [4, 2, 12, 16, 3, 6, 19, 19, 15, 1, 15, 13]
   El arreglo ordenado 2 es: [1, 2, 3, 4, 6, 12, 13, 15, 15, 16, 19, 19]

-> Arreglos con mayor tiempo de ejecución
   El arreglo original 2 es: [4, 2, 12, 16, 3, 6, 19, 19, 15, 1, 15, 13]
   El arreglo ordenado 2 es: [1, 2, 3, 4, 6, 12, 13, 15, 15, 16, 19, 19]
   El arreglo original 4 es: [19, 8, 17, 6, 15, 16, 4, 2, 7, 19, 2]
   El arreglo ordenado 4 es: [2, 2, 4, 6, 7, 8, 15, 16, 17, 19, 19]
   El arreglo original 3 es: [19, 12, 19, 1, 9, 11, 12, 18, 2, 10, 17, 6, 11, 10]
   El arreglo ordenado 3 es: [1, 2, 6, 9, 10, 10, 11, 11, 12, 12, 17, 18, 19, 19]
2023-10-24 21:53:43.133 Python[3540:227572] WARNING: Secure coding is automatically e
```

Figura 2: Prueba de la modificacion del pivote.

3. Codigo completo

Listing 13: Código completo

```
#Libreria para el tiempo
import time
#Libreria para poner datos aleatorios
import random
#Libreria para crear la grafica
import matplotlib.pyplot as plt

#Funcion que genera el tama o de el arreglo
def Tam():
    #Genera el tama o aleatoriamente
    t=random.randrange(5,15)
    return t

#Funcion que genera el arreglo
def GenA(n):
    #Llena el arreglo aleatoriamente
    A=[random.randrange(1,20) for i in range(n)]
    return A

#Funcion que genera una lista donde se guardan los 100 arreglos
def Arreglos():
    #Genera la lista vacia
    a=[]
    #Hace que nos genere tantos arreglos como queramos
    for i in range(100):
        #Va agregando a la lista cada arreglo generado
        a.append(GenA(Tam()))
    return a

def Media(x):
    #Usa sum, nos ayuda a sumar todos los elementos de un arreglo
    #La doble linea redondea la division a un numero entero
    #Len calcula la longitud del arreglo
    m=sum(x)//len(x)
    return m

def QuickSort(x):
    #Se define el caso base, que dice que cuando un arreglo solo tenga
        un elemento pues ya estara ordenado
    if len(x)<=1:
        return x
    #Se define el pivote, que ya se calculo anteriormente en la
       funcion media
    m=Media(x)
    p=m
    #Se definen los arreglos donde clasificaremos los elementos luego
       de las comparaciones
    izq=[]
    der=[]
    igual=[]
```

```
48      #Cilco que asigna cada valor del arreglo original a los arreglos
            resultantes de la divison
49      for e in x:
50          if e<p:
51              izq.append(e)
52          elif e>p:
53              der.append(e)
54          else:
55              #Se crea tambien un arreglo igual para el caso en que haya
                    numeros repetidos
56              #O iguales al pivote
57              igual.append(e)
58      #Se definen los dos arreglos en una variable al momento de
            volverlas a llamar
59      izq_ordenado=QuickSort(izq)
60      der_ordenado=QuickSort(der)
61      #Se regresan los arreglos resultantes de las divisiones del
            arreglo original
62      return izq_ordenado + igual + der_ordenado
63
64  #Funcion para calcular el tiempo de ejecucion de una funcion
65  def TiempoEjecucion(func, *args):
66      #Tomamos el tiempo de ejecucion incial
67      inicio=time.time()
68      #Llama a la funcion de la que queremos el tiempo de ejecucion
69      func(*args)
70      #Tomamos el tiempo de ejecucion final
71      fin=time.time()
72      #Calculamos el tiempo de ejecucion real
73      TiempoReal=fin-inicio
74      #Regresamos como resultado de la funcion el tiempo de ejecucion
75      return TiempoReal
76
77  print("\n\t\t\tQuickSort")
78  n=Tam()
79  #Lista que contiene cada arreglo
80  x=Arreglos()
81
82  #Imprime cada elemento de la lista
83  print("\n-> Arreglos generados aleatoriamente")
84  Tama os=[]
85  for i in x:
86      print(f"   El arreglo {x.index(i)} es: {i}")
87      Tama os.append(len(i))
88
89  #Manda cada arreglo a la funcion QuickSort para que ordene de uno por
        uno
90  print("\n-> Arreglos ordenados")
91  ArregloOrdenado=[]
92  for i in x:
93      m=Media(i)
94      #Cada arreglo ordenado lo guarda en una variable
```

```
95      ordenado=QuickSort(i)
96      ArregloOrdenado.append(ordenado)
97      #Imprime cada arreglo uno por uno ordenado
98      print(f"   El arreglo {x.index(i)} ordenado es: {ordenado}")
99
100 #Manda cada ejecucion de la funcion QuickSort
101 print("\n-> Tiempos de ejecucion")
102 Tiempos=[]
103 Resultados=[]
104 for i in x:
105     #Cada tiempo de ejecucion lo guardaremos en la variable tiempo
106     Tiempo=TiempoEjecucion(QuickSort,i)
107     #Imprime el tiempo de ejecucion de cada arreglo generado
108     print(f"   Tiempo tardado en ordenar el arreglo {x.index(i)}: {
            Tiempo}")
109     #Se guarda cada tiempo de ejecucuion en un arreglo
110     Tiempos.append(Tiempo)
111     Resultados.append((Tiempo, i, QuickSort(i)))
112
113 Resultados = sorted(Resultados, key=lambda x: x[0])
114
115 print("\n-> Arreglos con menor tiempo de ejecuci n")
116 for i, (tiempo, arreglo_original, arreglo_ordenado) in enumerate(
        Resultados[:3], 1):
117     print(f"   El arreglo original {x.index(arreglo_original)} es: {
            arreglo_original}")
118     print(f"   El arreglo ordenado {x.index(arreglo_original)} es: {
            arreglo_ordenado}")
119
120 print("\n-> Arreglos con mayor tiempo de ejecuci n")
121 for i, (tiempo, arreglo_original, arreglo_ordenado) in enumerate(
        Resultados[-3:], 1):
122     print(f"   El arreglo original {x.index(arreglo_original)} es: {
            arreglo_original}")
123     print(f"   El arreglo ordenado {x.index(arreglo_original)} es: {
            arreglo_ordenado}")
124
125 plt.xlabel('Tama o del arreglo (n)')
126 plt.ylabel('Tiempo de ejecuci n (segundos)')
127 plt.title('Gr fico del algoritmo QuickSort')
128
129 # Usamos n como el tama o del arreglo en el eje X y Tiempos en el eje
        Y
130 plt.plot(Tiempos)
131 plt.show()
```

4. Resultados (Capturas)

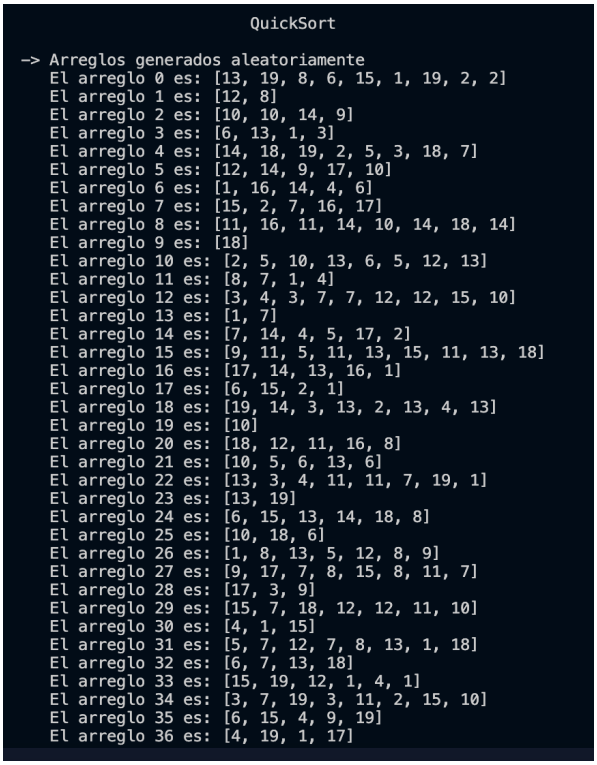

Figura 3: Casos de prueba (Parte 1)

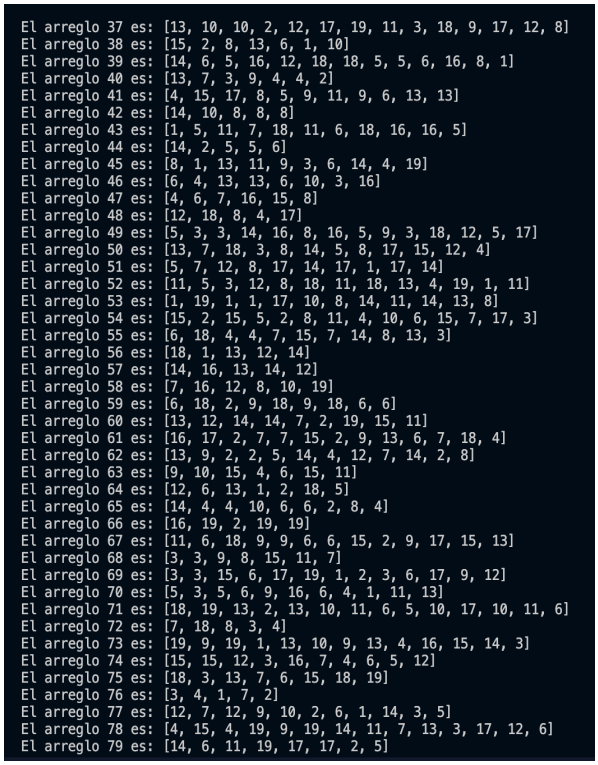

Figura 4: Casos de prueba (Parte 2)

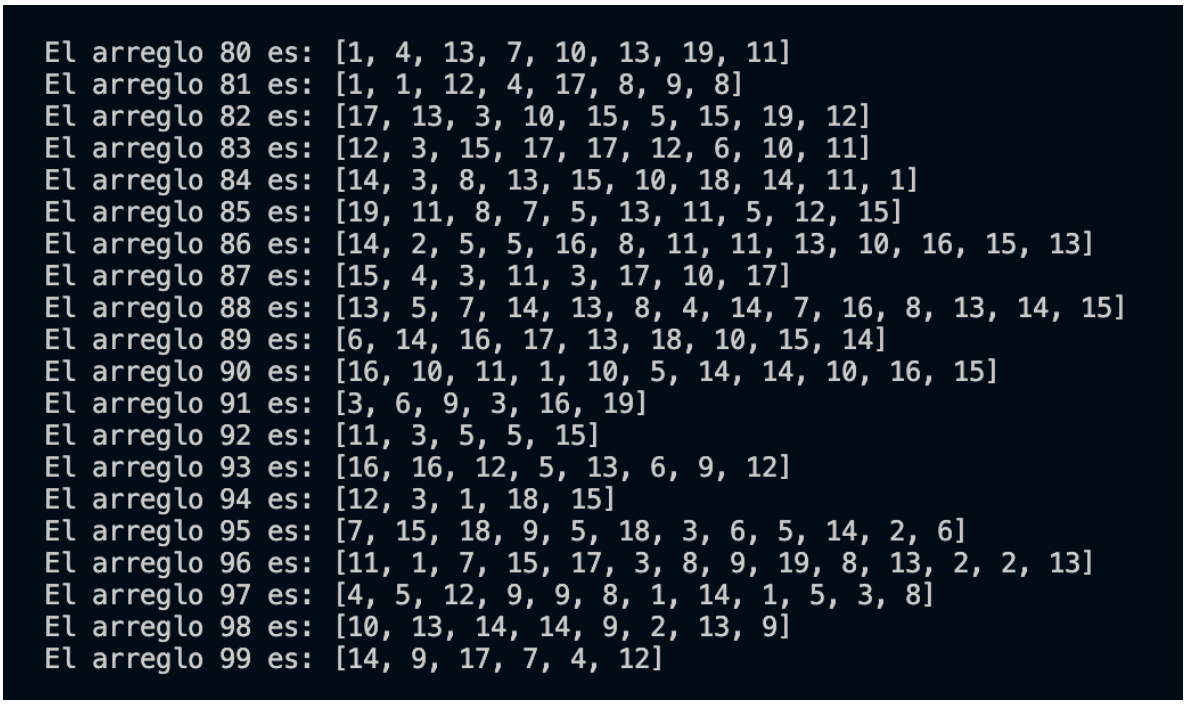

Figura 5: Casos de prueba (Parte 3)

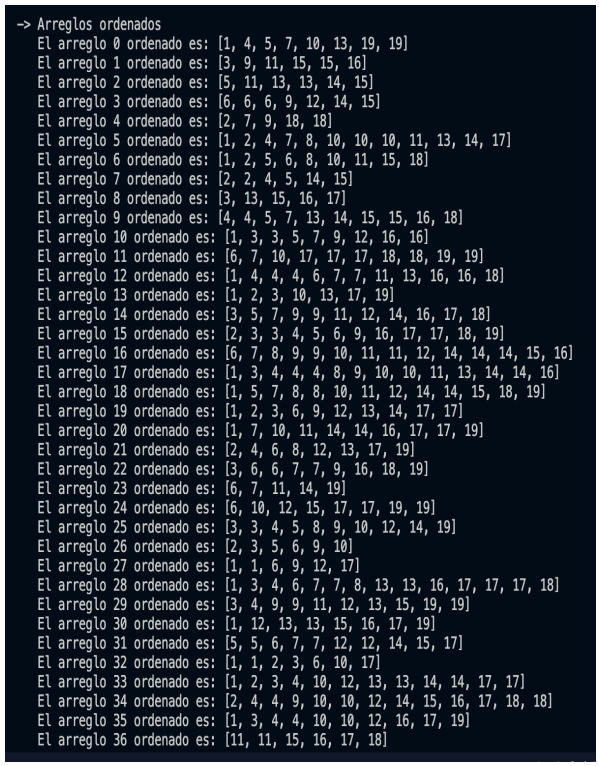

Figura 6: Arreglos ordenados (Parte 1)

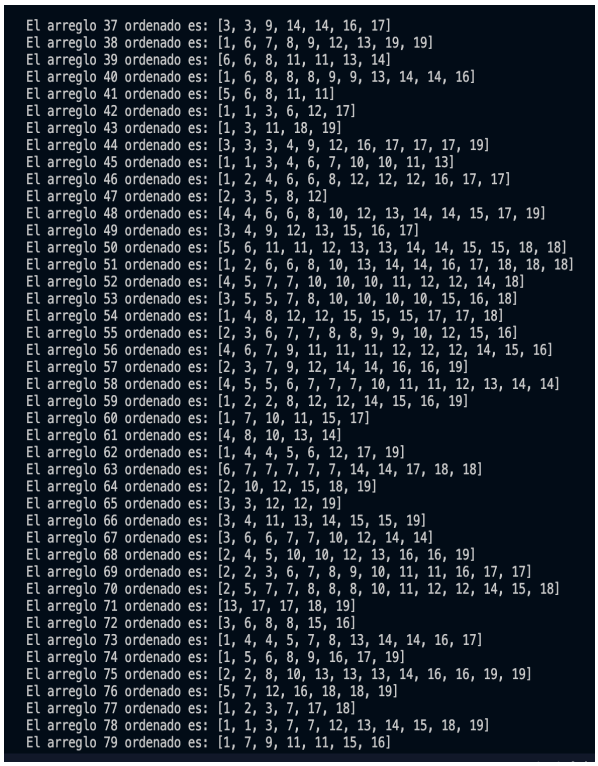

Figura 7: Arreglos ordenados (Parte 2)

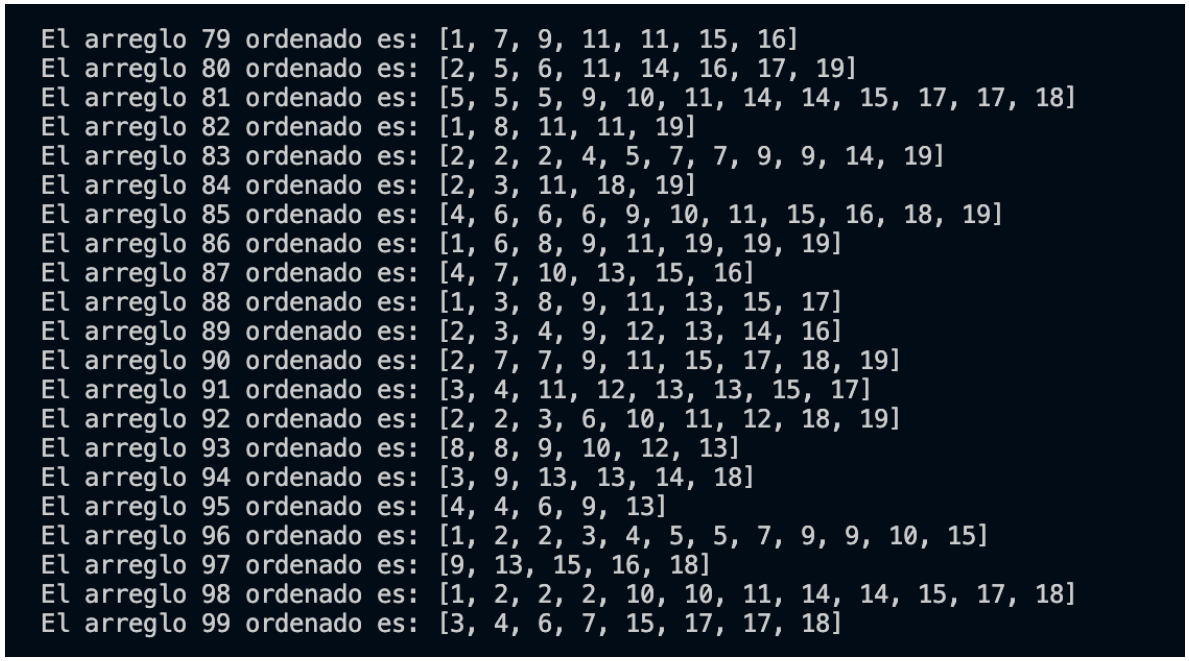

Figura 8: Arreglos ordenados (Parte 3)

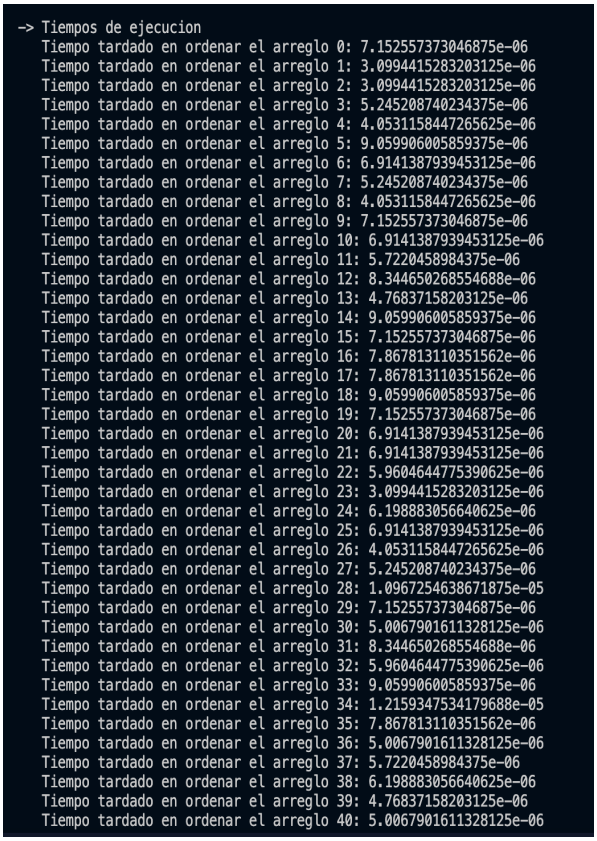

Figura 9: Tiempos de ejecución (Parte 1)

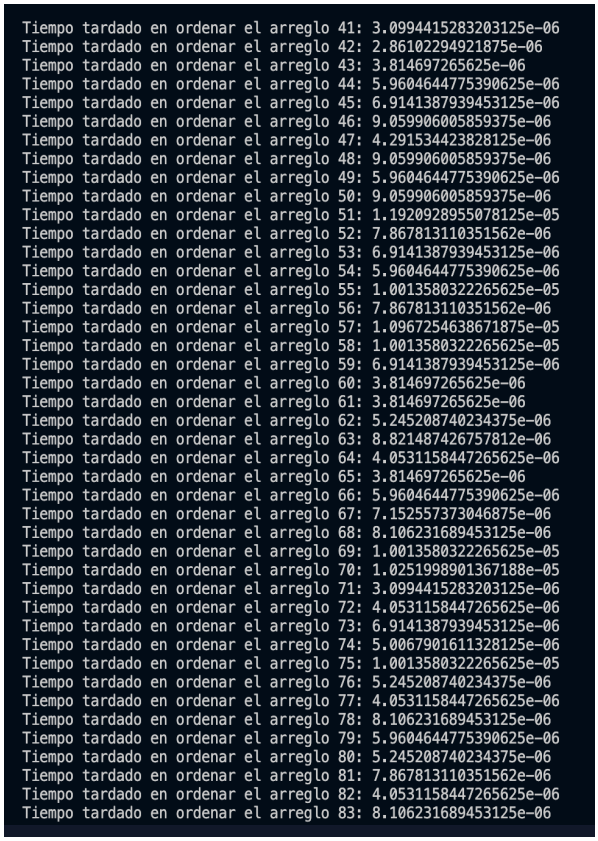

Figura 10: Tiempos de ejecución (Parte 2)

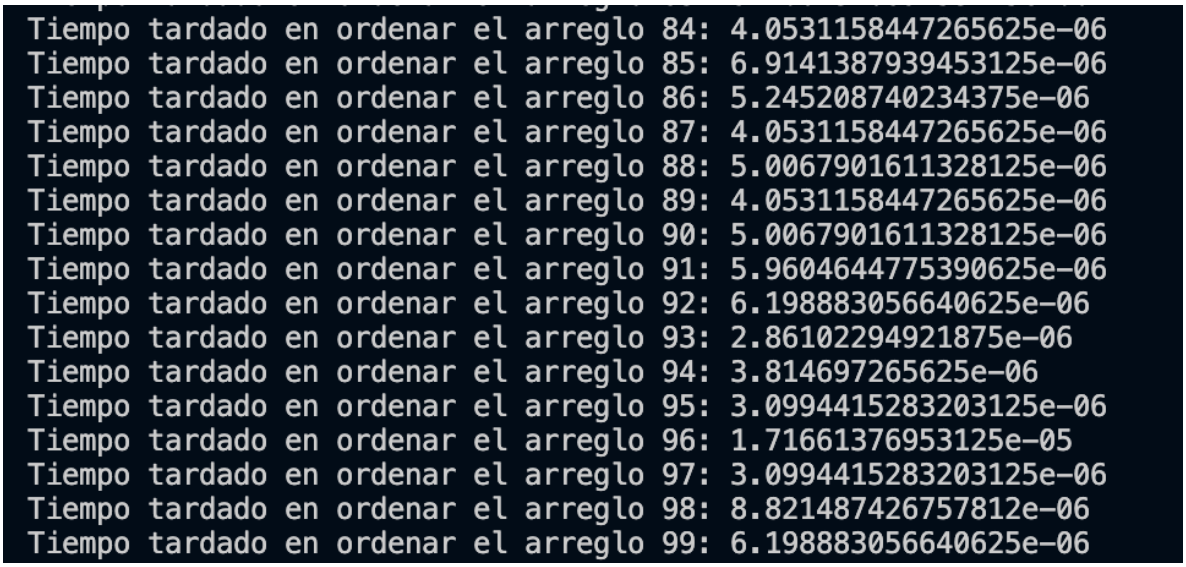

Figura 11: Tiempos de ejecución (Parte 3)

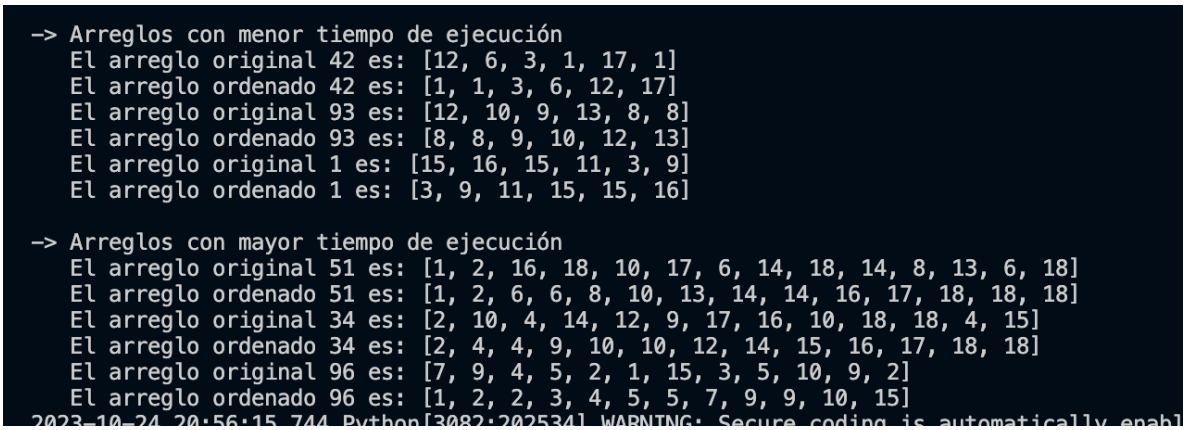

Figura 12: Arreglos con mas y menos tiempo de ejecución

5. Conclusiones

Gracias a la práctica, analizamos y comprendimos a fondo el funcionamiento del algoritmo Quick-Sort. Del mismo modo, entendimos por qué es más rápido que los demás, dado que a pesar de que proporcionábamos distintos tamaños de algoritmos, el tiempo de ejecución no cambiaba de manera drástica, y el tiempo, a pesar de ser mayor que el de los arreglos con menos elementos, tampoco aumentaba demasiado.

También, en lo personal, practiqué mucho más la sintaxis de Python y comprendí de mejor manera las funciones y la utilidad de estas, y se me hizo mucho más fácil usar funciones que haberlo hecho todo dentro del ámbito del código.

6. Referencias

- https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-iv/2019/ii/guia-4.pdf
- https://numerentur.org/quicksort/
- https://www.genbeta.com/desarrollo/implementando-el-algoritmo-quicksort
- http://aniei.org.mx/paginas/uam/CursoAA/curso_aa_19.html#:~:text=El%20mejor%20caso%20de%20quick,lista%20original%20en%20dos%20listas.