| | |
|---|---|
| | **Carátula para entrega de prácticas** |
| Facultad de Ingeniería | Laboratorio de docencia |

# Laboratorios de computación salas A y B

*Profesor:* Jesús Cruz Navarro

*Asignatura:* Estructuras de Datos y Algoritmos 2

*Grupo:* 1

*No de Práctica(s):* 2

*Integrante(s):* Ugalde Velasco Armando

*No. de Equipo de cómputo empleado:*

*No. de Lista o Brigada:* 32

*Semestre:* 2021-1

*Fecha de entrega:* 14 de octubre de 2020

*Observaciones:*

CALIFICACIÓN: _____

# PRÁCTICA 2: ORDENAMIENTO 2

**Objetivo:** El estudiante conocerá e identificará la estructura de los algoritmos de ordenamiento *QuickSort* e *InsertionSort*.

**1. Programar los algoritmos de ordenamiento vistos en clase. Cada algoritmo debe estar en una función y recibir solamente un arreglo de datos.**

**a**. *QuickSort*

```python
def partition(arr, start, end):
    pivot = arr[end]
    i = start - 1
    for j in range(start, end):
        if arr[j] ⩽ pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[end] = arr[end], arr[i + 1]
    return i + 1


def quickSortHelper(arr, start, end):
    if start < end:
        pivot = partition(arr, start, end)
        quickSortHelper(arr, start, pivot - 1)
        quickSortHelper(arr, pivot + 1, end)


def quickSort(arr):
    quickSortHelper(arr, 0, len(arr) - 1)
```

**Implementación de QuickSort**

Se implementó el algoritmo analizado en clase, agregando una función auxiliar (**quickSortHelper**), para que la función **quickSort** sólo recibiera el arreglo a ordenar como argumento.

Como se analizó en clase, el funcionamiento del algoritmo consiste en particionar el arreglo utilizando la subrutina **partition**, escogiendo el último elemento como el pivote y

retornando el índice de su posición después de haber realizado la partición. Finalmente, se ordenan recursivamente las mitades del arreglo correspondientes. La primera condición en la función **quickSortHelper** comprueba la existencia de al menos dos elementos en el arreglo, de lo contrario, la función retorna sin realizar ninguna operación.

Como ya se mencionó en el análisis previo del algoritmo, la complejidad asintótica de su caso promedio es **O(n log n),** y en los peores casos es **O($n^2$).**

**b.** *InsertionSort*

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        currentElement = arr[i]
        j = i - 1
        while j ⩾ 0 and arr[j] > currentElement:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = currentElement
```

**Implementación de InsertionSort**

La función **insertionSort** ordena el arreglo proporcionado. Su funcionamiento, ya analizado en clase, consiste en corregir las inversiones que provoca cada elemento. Es decir, en cada iteración se tiene la invariante de que los elementos en la parte izquierda del elemento analizado ya se encuentran ordenados, por lo tanto, se corrigen las posibles inversiones producidas por el elemento presente en el índice analizado. Al terminar el ciclo exterior, todos los elementos del arreglo se encuentran ordenados, tomando como argumento la invariante antes mencionada.

La complejidad de este algoritmo es **O($n^2$)** para el caso promedio *(lista aleatoria)* y el peor caso *(lista ordenada de forma inversa),* sin embargo, para el mejor caso *(lista ordenada)* su complejidad es **O(n)**, ya que solamente se realiza un recorrido en el arreglo donde el ciclo interno nunca se ejecuta.

**2. Desarrollar un programa para probar los algoritmos implementados anteriormente y registrar sus tiempos de ejecución con listas de tamaño n. Para probar, ambos algoritmos deben recibir el mismo conjunto de elementos (es decir, si es una lista aleatoria, los algoritmos deben recibir el mismo conjunto aleatorio). El programa debe imprimir al final n y el tiempo de ejecución de cada algoritmo.**

**Se debe probar con listas de diferentes tamaños (n= 500, 1000, 2000, 5000) y registrar los tiempos de ejecución para cada uno de los siguientes casos:**

**a.** *Lista Ordenada Ascendente*
**b.** *Lista Ordenada Descendente*
**c.** *Lista Aleatoria*

**3. Dado que los tiempos pueden variar de una ejecución a otra, realice las pruebas 3 veces y obtenga el promedio de los tiempos. Nota 1: Para generar lista ordenada ascendentemente, puede agregue el valor de i del ciclo for a la lista. Para generar lista ordenada descendentemente, puede usar for i in range( max, min, - 1), lo hará que i tome los valores de max hasta min (exclusivo) en cada iteración. Nota 2: Si hay problemas con el límite de recursión que usa por default Python, modifíquelo para permitir los peores casos de QuickSort.**

El programa realizado consta de distintas funciones auxiliares, las cuales se mostrarán a continuación.

Primero, se definieron algunas constantes importantes: los algoritmos a utilizar, sus nombres, y los valores mínimo y máximo de los elementos en los arreglos a ordenar. Además, se definió una lista con los valores de **n** a analizar y se modificó el límite de recursión a **6000** para lograr realizar las pruebas correspondientes de los casos más lentos de **quickSort**.

```
ALG1 = quickSort.quickSort
ALG2 = insertionSort.insertionSort
ALG1_NAME = "QuickSort"
ALG2_NAME = "InsertionSort"

sys.setrecursionlimit(6000)

# Valores mínimo y máximo de los el
MIN_VALUE = -100000
MAX_VALUE = 100000

# Valores de n a analizar
X_ARR = [500, 1000, 2000, 5000]
```

**Constantes a utilizar en el programa**

### sortAndComputeTime

Esta función ordena una lista con el algoritmo proporcionado **(función sortingAlg)**, y calcula el tiempo utilizado por éste.

```python
def sortAndComputeTime(sortingAlg, arr):
    # Mantener una copia ordenada
    sortedArr = arr.copy()
    sortedArr.sort()

    # Realizar una copia para evitar mutaciones en el arreglo original
    copy = arr.copy()

    # Ordenar la lista y calcular su tiempo de ejecución
    t1 = time.time()
    sortingAlg(copy)
    t2 = time.time()

    # Comprobar que el algoritmo ordenó correctamente la lista
    assert copy == sortedArr

    # Devolver el tiempo utilizado
    return t2 - t1
```

**Función sortAndComputeTime**

### sortAndGetTimes

Esta función ordena la misma lista con los dos algoritmos de ordenamiento analizados y retorna una tupla con los tiempos utilizados por cada uno.

```python
def sortAndGetTimes(arr):
    print("N =", len(arr))

    firstAlgTime = sortAndComputeTime(ALG1, arr)
    print(ALG1_NAME, ":", firstAlgTime)

    secondAlgTime = sortAndComputeTime(ALG2, arr)
    print(ALG2_NAME, ":", secondAlgTime, "\n")

    return firstAlgTime, secondAlgTime
```

**Función sortAndGetTimes**

### computeAvgTimesArr

Esta función retorna una lista de tuplas con los tiempos promedio de ordenamiento para cada **n**, utilizando el generador de listas proporcionado. Éste último es el que determina el tipo de caso presentado (mejor, peor o promedio).

```python
def computeAvgTimesArr(listGenerator):
    # Lista con las tuplas de tiempos promedios para cada n.
    AVG_TIMES = []
    # Calcular los tiempos promedio para cada n y agregar la tupla respectiva a l
    for n in X_ARR:
        # Lista que contendrá 3 tuplas con los tiempos obtenidos para n
        TIMES_FOR_N = []
        # Calcular 3 veces el tiempo de los algoritmos, ordenando
        # listas diferentes en cada iteración (utilizando el generador dado)
        for i in range(3):
            arr = listGenerator(n)
            TIMES_FOR_N.append(sortAndGetTimes(arr))

        # Calcular el promedio de tiempo para cada algoritmo
        firstMean = numpy.mean([t[0] for t in TIMES_FOR_N])
        print("Tiempo promedio", ALG1_NAME, "para", n, "elementos:", firstMean)
        secondMean = numpy.mean([t[1] for t in TIMES_FOR_N])
        print("Tiempo promedio", ALG2_NAME, "para", n, "elementos:", secondMean,
        # Agregar la tupla de tiempo promedio obtenida
        AVG_TIMES.append((firstMean, secondMean))
    return AVG_TIMES
```

**Función computeAvgTimesArr**

### testAndPlot

Esta función obtiene los tiempos promedios para cada **n** utilizando el generador de listas proporcionado, e imprime la gráfica con la información necesaria utilizando la librería *matplotlib*. De nuevo, **listGenerator** determina el caso a analizar, y el parámetro **title** debe contener el título del caso a mostrar en la gráfica.

```python
def testAndPlot(listGenerator, title):
    print(title, "\n")
    AVG_TIMES = computeAvgTimesArr(listGenerator)

    fig, ax = plt.subplots()
    ax.set_title(title)
    ax.set_xlabel("Tamaño del arreglo")
    ax.set_ylabel("Tiempo de ejecución")
    ax.plot(X_ARR, [y[0] for y in AVG_TIMES])
    ax.plot(X_ARR, [y[1] for y in AVG_TIMES])
    ax.legend([ALG1_NAME, ALG2_NAME])
```

**Función testAndPlot**

### Funciones para cada caso

Estas funciones llevan a cabo los tests correspondientes de rendimiento para cada algoritmo, y grafican sus respectivos resultados. Nótese que, dentro de cada función, se encuentran definidas las funciones generadoras de listas para cada caso: en el caso promedio, se retorna una lista aleatoria; en el mejor caso, se retorna una lista aleatoria ordenad; y finalmente, en el peor caso se retorna una lista aleatoria ordenada de forma inversa. Como se mencionó anteriormente, éstas funciones determinan las características de los casos analizados.

```python
def mejorCaso():
    # Devuelve una lista aleatoria ordenada de tamaño n
    def listGenerator(n):
        arr = [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]
        arr.sort()
        return arr

    testAndPlot(listGenerator, "MEJOR CASO")
```

```python
def casoPromedio():
    # Devuelve una lista aleatoria de tamaño n
    def listGenerator(n):
        return [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]

    testAndPlot(listGenerator, "CASO PROMEDIO")
```

```python
def peorCaso():
    # Devuelve una lista aleatoria ordenada de forma inversa, de tamaño n
    def listGenerator(n):
        arr = [random.randint(MIN_VALUE, MAX_VALUE) for j in range(n)]
        arr.sort()
        arr.reverse()
        return arr

    testAndPlot(listGenerator, "PEOR CASO")
```

**Funciones para realizar los tests de cada caso**

Finalmente, se ejecutan las funciones anteriores y se muestran las gráficas respectivas:

```python
casoPromedio()
mejorCaso()
peorCaso()
plt.show()
```

**Ejecución de los tests y graficación**

A continuación, se muestran algunas capturas de pantalla de la salida producida por la ejecución de los tres tests:

```
CASO PROMEDIO

N = 500
QuickSort : 0.0009555816650390625
InsertionSort : 0.010970830917358398

N = 500
QuickSort : 0.0009913444519042969
InsertionSort : 0.011968851089477539

N = 500
QuickSort : 0.0009717941284179688
InsertionSort : 0.011022567749023438

Tiempo promedio QuickSort para 500 elementos: 0.000972906748453776
Tiempo promedio InsertionSort para 500 elementos: 0.011320749918619791

N = 1000
QuickSort : 0.003016948699951172
InsertionSort : 0.0508420467376709

N = 1000
QuickSort : 0.002991199493408203
InsertionSort : 0.048868656158447266

N = 1000
QuickSort : 0.0029420852661132812
InsertionSort : 0.05385422706604004
```

**Salida del programa**

```
MEJOR CASO

N = 500
QuickSort : 0.04396462440490723
InsertionSort : 0.0

N = 500
QuickSort : 0.051877498626708984
InsertionSort : 0.000997781753540039

N = 500
QuickSort : 0.0329132080078125
InsertionSort : 0.0

Tiempo promedio QuickSort para 500 elementos: 0.04291844367980957
Tiempo promedio InsertionSort para 500 elementos: 0.0003325939178466797

N = 1000
QuickSort : 0.12267160415649414
InsertionSort : 0.0

N = 1000
QuickSort : 0.11920976638793945
InsertionSort : 0.0009953975677490234

N = 1000
QuickSort : 0.11905384063720703
InsertionSort : 0.0009911060333251953

Tiempo promedio QuickSort para 1000 elementos: 0.12031173706054688
```

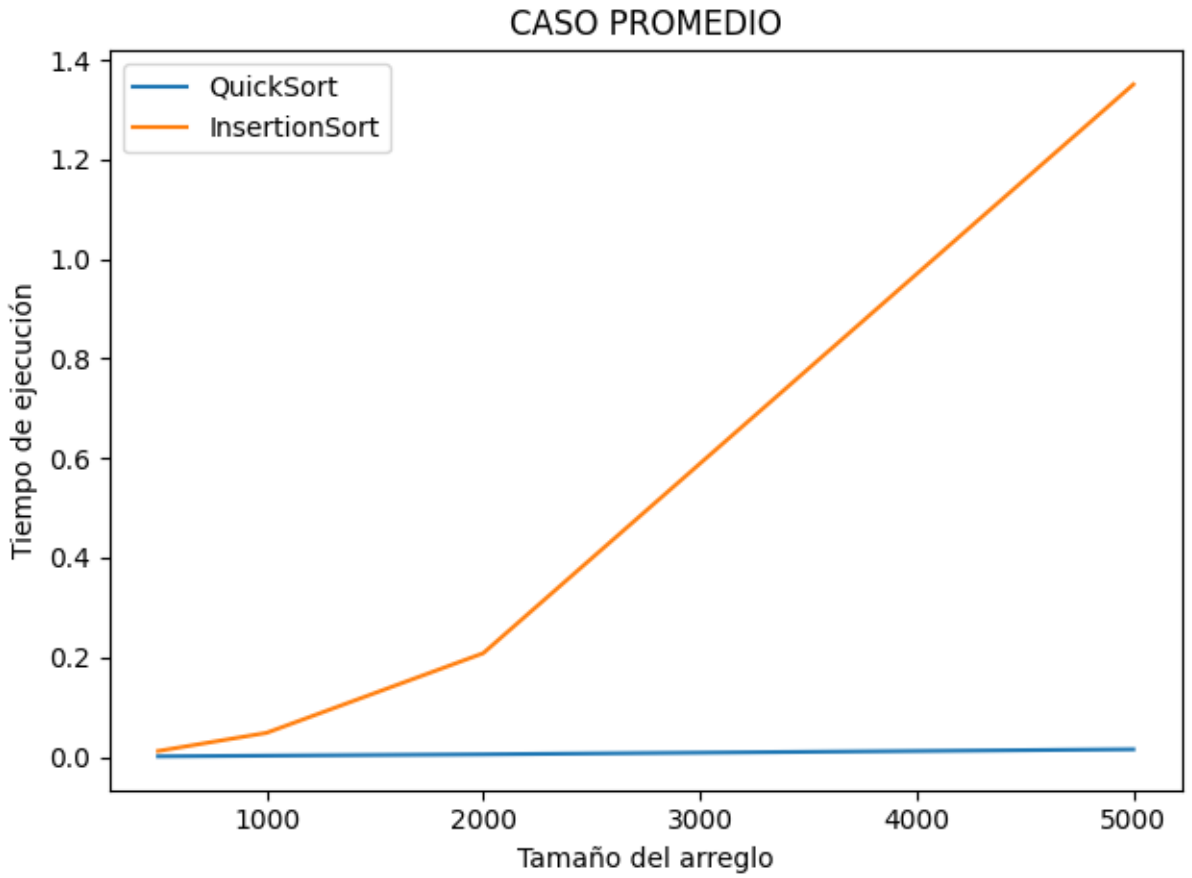
**Salida del programa**

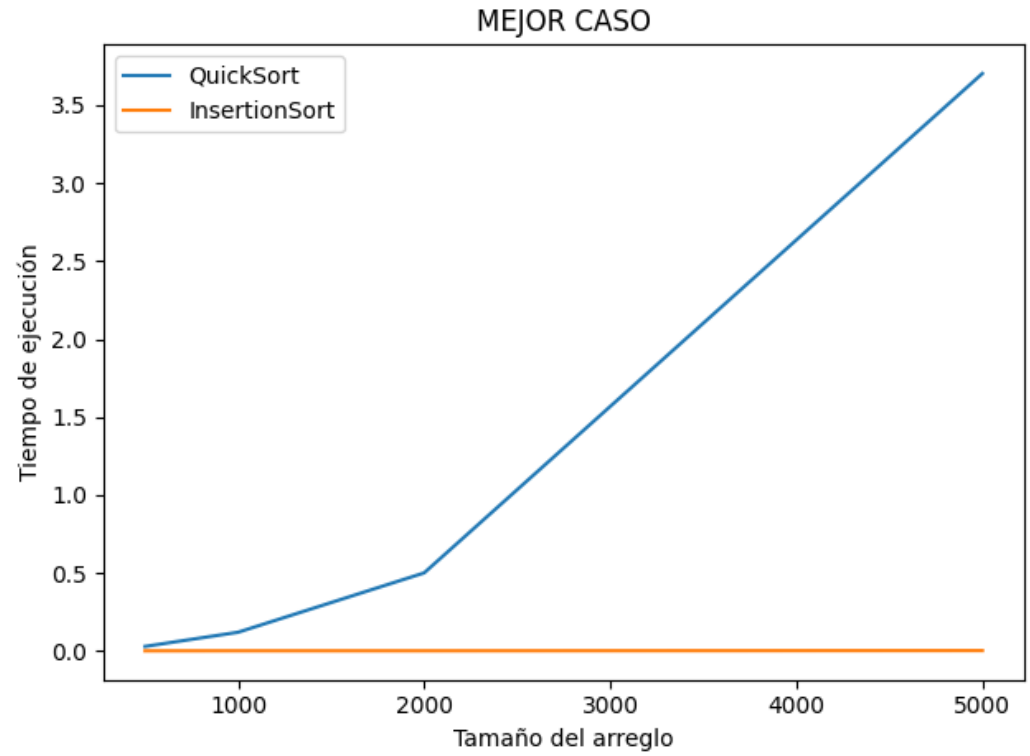
```
PEOR CASO

N = 500
QuickSort : 0.036106109619140625
InsertionSort : 0.04484224319458008

N = 500
QuickSort : 0.018950700759887695
InsertionSort : 0.024497270584106445

N = 500
QuickSort : 0.018947601318359375
InsertionSort : 0.024935245513916016

Tiempo promedio QuickSort para 500 elementos: 0.024668137232462566
Tiempo promedio InsertionSort para 500 elementos: 0.031424919764200844

N = 1000
QuickSort : 0.1018369197845459
InsertionSort : 0.13376069068908691

N = 1000
QuickSort : 0.10073351860046387
InsertionSort : 0.10875606536865234

N = 1000
QuickSort : 0.10969686508178711
InsertionSort : 0.15454578399658203

Tiempo promedio QuickSort para 1000 elementos: 0.10408910115559895
```

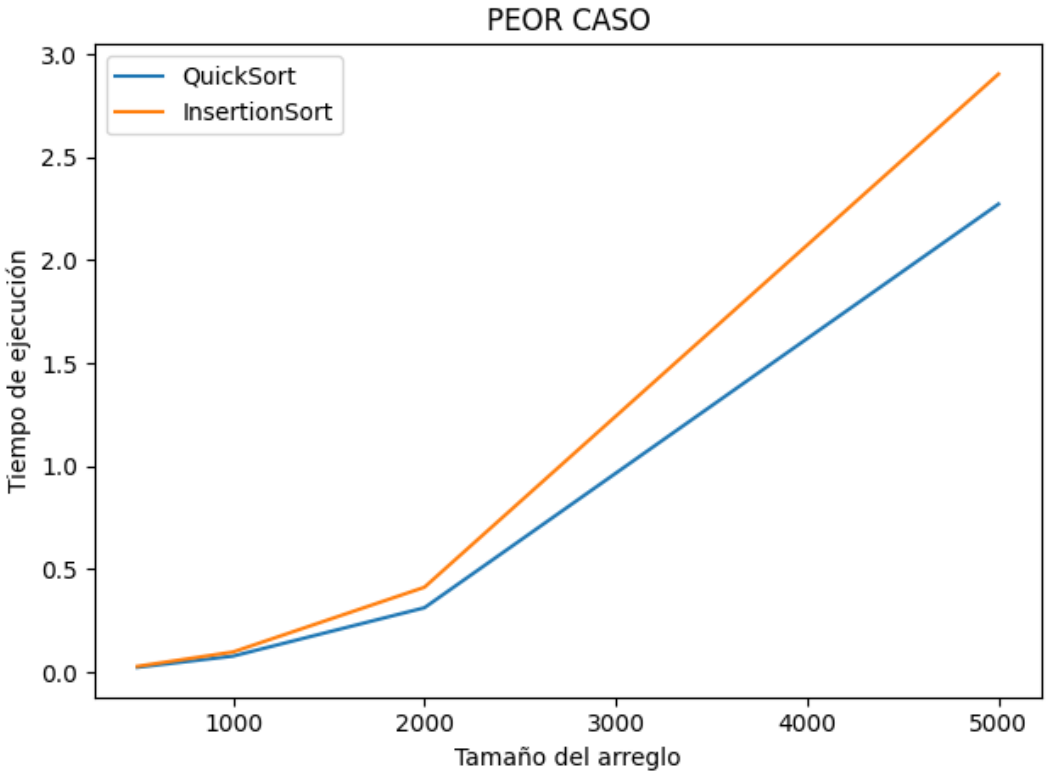
**Salida del programa**

**4. Genere una gráfica para cada caso (3 gráficas), para comparar los tiempos de ambos algoritmos en función del tamaño de la lista (n vs t, es decir, las n en el eje X y los tiempos en segundos en el eje Y).**

A continuación, se muestran las gráficas generadas por el programa anterior:






**Gráficas de rendimiento para cada caso**

Como se logra apreciar, en el caso promedio, los tiempos de ejecución de **insertionSort** superaron por mucho a los de **quickSort**. Esto comprueba la complejidad correspondiente mencionada anteriormente para cada algoritmo. De hecho, la forma de curva correspondiente a **insertionSort** es parecida a la de una función cuadrática, tal como se esperaba.

Por otro lado, en el mejor caso *(lista ordenada)*, el tiempo de ejecución de **quickSort** fue mucho mayor al de **insertionSort**. Lo anterior se debe al caso límite comentado en clase respecto al algoritmo **quickSort**: el pivote elegido siempre termina en el último lugar del arreglo, por lo tanto, se realizan **n – 1** particiones y la complejidad asintótica del algoritmo se vuelve cuadrática. En el caso de **insertionSort**, como se comentó anteriormente, únicamente se realiza un recorrido en el arreglo, ya que no hay inversiones que corregir, por lo tanto, su complejidad es lineal. Lo anterior se ve reflejado claramente en la gráfica: la curva correspondiente a **quickSort** tiene una forma similar a la de una función cuadrática.

Finalmente, en el peor caso *(lista ordenada de forma inversa)*, los dos algoritmos presentan un rendimiento similar: las formas de sus gráficas son muy parecidas a la de una función cuadrática, sin embargo, **insertionSort** parece tener tiempos de ejecución mayores, por lo tanto, se puede inferir que sus factores constantes son mayores que los de la función correspondiente al rendimiento de **quickSort**. El comportamiento de **quickSort** en este caso es similar al del mejor caso, ya que las particiones se realizan de forma muy dispar, es decir, el pivote elegido siempre termina en la parte inicial del arreglo al momento de particionarlo. Por otro lado, notamos que, al estar ordenado de forma inversa, el arreglo tiene $\mathbf{n^2}$ inversiones, lo cual es directamente proporcional al número de operaciones que **insertionSort** realiza, determinando su complejidad.

## CONCLUSIONES

**QuickSort** e **InsertionSort** son dos algoritmos de ordenamiento que, si bien resuelven el mismo problema, tienen complejidades asintóticas distintas para el caso promedio: **O(n log n)** y **O($n^2$)** y respectivamente, como se determinó en clase. Como es evidente, el rendimiento de **quickSort** es mucho mejor que el de **insertionSort** en este caso. La complejidad asintótica de **quickSort** es, al igual que la de **mergeSort**, óptima para algoritmos de ordenamiento basados en comparación, lo que lo hace una excelente opción para realizar dicha tarea.

Sin embargo, para los casos donde se requiere ordenar una lista ya ordenada u ordenada de forma inversa, la complejidad de **quickSort** se torna cuadrática, representando una gran desventaja. Para resolver el problema anterior, es posible tomar nuevas estrategias al escoger el pivote. Una de las más comunes y eficientes es la de escogerlo aleatoriamente: de esta forma, un posible adversario no podría dar alguna entrada "maliciosa" en específico con el objetivo de que el rendimiento del algoritmo fuera malo. En este caso, el rendimiento esperado del algoritmo sería el del caso promedio, es decir, su complejidad asintótica sería **O(n log n).**

Por otro lado, si bien **insertionSort** es un algoritmo con una complejidad cuadrática, su simplicidad podría representar una ventaja para su comprensión. Además, para el mejor caso únicamente realiza un recorrido sobre el arreglo, por lo tanto, su rendimiento sería mejor pero únicamente en este caso. De hecho, el rendimiento de este algoritmo depende del numero de inversiones presentes en el arreglo.

El rendimiento de **quickSort**, como ya se mencionó, es óptimo en su caso promedio, por lo tanto, su utilización es viable para la mayoría de las aplicaciones. Sin embargo, al tener una naturaleza no determinística, es posible tener casos extremos que podrían afectar considerablemente el funcionamiento del sistema donde se utilice.