



## **Análisis de Algoritmos.**

### **Tarea #1.**

#### **Profesor:**

**Joss Rayn Pecou Johnson.**

#### **Estudiantes:**

**Deiner Céspedes Molina.**

**Esteban Rodríguez Vargas.**

**Yader Siezar Chaves.**

**Semestre II, Año 2025.**

## Parte 1:

### Explicación del algoritmo:

Decidimos usar el algoritmo de ordenamiento QuickSort como el principal para esta parte, en la cual los pasos son los siguientes:

- Se elige el último elemento como pivote.
- Se divide la lista en dos sublistas (menores y mayores).
- Se llama recursivamente a quicksort sobre cada sublista.
- Se concatenan los resultados.

### Análisis de complejidad temporal:

$$T(n) = 1 + 1 + n\log(n) + n\log(n) + 1 + 1 + 1$$

$$T(n) = 2n\log(n) + 5$$

$$\text{Mejor caso: } T(n) = \Omega(n\log n)$$

$$\text{Peor caso: } T(n) = O(n^2)$$

$$\text{Caso promedio: } T(n) = \Theta(n\log n)$$

Podemos observar que el peor caso sucede cuando el pivote es el número mayor o menor de toda la lista, si eso pasa la partición queda desbalanceada, ya que una sublista queda con todos los elementos menos el pivote y la otra queda vacía, e iría así elemento a elemento hasta quedar con una lista de tamaño 1. Esto que provocaría una especie de bucle que haría el proceso de ordenamiento el doble de extenso de lo normal.

## Resultados experimentales:

Se muestran los resultados de pruebas con nuestro algoritmo (QuickSort).

Imágenes:

```
===== Tamaño: 10x10 ===== 100 elementos  
QuickSort -> Tiempo: 0.0011508464813232422 Memoria: 0.00240325927734375 MB
```

```
===== Tamaño: 32x32 ===== 1024 elementos  
QuickSort -> Tiempo: 0.01181650161743164 Memoria: 0.01158905029296875 MB
```

```
===== Tamaño: 100x100 ===== 10000 elementos  
QuickSort -> Tiempo: 0.13506579399108887 Memoria: 0.08806610107421875 MB
```

```
===== Tamaño: 317x317 ===== 100489 elementos  
QuickSort -> Tiempo: 1.2995517253875732 Memoria: 0.7979660034179688 MB
```

```
===== Tamaño: 1000x1000 ===== 1000000 elementos  
QuickSort -> Tiempo: 13.096932649612427 Memoria: 7.7234649658203125 MB
```

### Comparación con otro algoritmo:

En las siguientes imágenes se observan los resultados de la comparación entre los métodos QuickSort y MergeSort en una matriz de tamaño  $n \times n$ . Podemos observar cómo los resultados son muy similares en la mayoría de los casos tanto en la memoria utilizada como en el tiempo de ejecución.

Imágenes:

```
===== Tamaño: 10x10 ===== 100 elementos
QuickSort -> Tiempo: 0.001058816909790039 Memoria: 0.00262451171875 MB
MergeSort -> Tiempo: 0.001203775405883789 Memoria: 0.0016937255859375 MB
```

```
===== Tamaño: 50x50 ===== 2500 elementos
QuickSort -> Tiempo: 0.0285336971282959 Memoria: 0.0258941650390625 MB
MergeSort -> Tiempo: 0.02178668975830078 Memoria: 0.02275848388671875 MB
```

```
===== Tamaño: 300x300 ===== 90000 elementos
QuickSort -> Tiempo: 1.256836175918579 Memoria: 0.7160873413085938 MB
MergeSort -> Tiempo: 0.8821108341217041 Memoria: 0.7249526977539062 MB
```

```
===== Tamaño: 1000x1000 ===== 1000000 elementos
QuickSort -> Tiempo: 13.02462887763977 Memoria: 7.740478515625 MB
MergeSort -> Tiempo: 15.553703546524048 Memoria: 8.460205078125 MB
```

## Parte 2:

### Explicación del algoritmo:

El algoritmo utilizado en este apartado fue el ya existente Búsqueda Hash, este algoritmo funciona de la siguiente manera:

- Se crea una lista con todas las posiciones de los elementos de la matriz.
- Se revisa si el elemento está en la lista y se retorna el resultado.

### Análisis de complejidad temporal:

$$T(n) = 1 + 1 + n*n + 1 + 1 + 1 + 1 + 1 + 1$$

$$T(n) = n^2 + 8$$

$$\text{Peor caso: } T(n) = O(n^2)$$

$$\text{Mejor caso: } T(n) = \Omega(n^2)$$

$$\text{Caso promedio: } T(n) = \theta(n^2)$$

Podemos observar que nuestro algoritmo siempre tendrá un orden de  $n^2$ , esto se debe a que la matriz siempre se va a recorrer de inicio a fin, lo que hace que a mayor número de elementos aumente el tiempo y el espacio que se consume en la ejecución.

## Resultados experimentales:

Se muestran los resultados de pruebas con nuestro algoritmo (Busqueda Hash).

Imágenes:

```
Tamaño de matriz: 10 x 10
Numero buscado: 39

Busqueda Hash:
No encontrado
Tiempo de ejecución: 0.00018978118896484375 segundos
Memoria usada: 4.8359375 KB
```

```
Tamaño de matriz: 100 x 100
Numero buscado: 39

Busqueda Hash:
Encontrado
Tiempo de ejecución: 0.0032372474670410156 segundos
Memoria usada: 10.4453125 KB
```

```
Tamaño de matriz: 500 x 500
Numero buscado: 39

Busqueda Hash:
Encontrado
Tiempo de ejecución: 0.10676908493041992 segundos
Memoria usada: 12.37890625 KB
```

## Comparación con otro algoritmo:

Al realizar comparaciones con el algoritmo de búsqueda lineal logramos observar los siguientes puntos:

1. La memoria que utiliza la búsqueda Hash aumenta basada en la entrada que se le da, mientras que la búsqueda lineal siempre utiliza la misma cantidad de memoria, ya que no guarda datos en la misma como lo hace Hash.
2. En todos los casos probados el algoritmo de búsqueda lineal requiere menos tiempo de ejecución que la búsqueda Hash, esto se debe a que la búsqueda lineal no siempre recorre la matriz completa mientras que nuestro algoritmo lo hace siempre.

Estos puntos se pueden evidenciar en las siguientes imágenes:

```
Tamaño de matriz: 10 x 10
Numero buscado: 39

Busqueda Hash:
Encontrado
Tiempo de ejecución: 0.0002830028533935547 segundos
Memoria usada: 5.1640625 KB

Busqueda lineal:
Encontrado
Tiempo de ejecución: 0.00016379356384277344 segundos
Memoria usada: 0.3984375 KB
```

```
Tamaño de matriz: 100 x 100
Numero buscado: 39

Busqueda Hash:
Encontrado
Tiempo de ejecución: 0.0024678707122802734 segundos
Memoria usada: 10.4453125 KB

Busqueda lineal:
Encontrado
Tiempo de ejecución: 0.00020933151245117188 segundos
Memoria usada: 0.3984375 KB
```

```
Tamaño de matriz: 500 x 500
Numero buscado: 39

Busqueda Hash:
Encontrado
Tiempo de ejecución: 0.10535931587219238 segundos
Memoria usada: 12.34765625 KB

Busqueda lineal:
Encontrado
Tiempo de ejecución: 0.00016808509826660156 segundos
Memoria usada: 0.3984375 KB
```

**Enlace del repositorio en el que se desarrolla la tarea:**

[https://github.com/ysiezar21/Tarea\\_corta\\_-1](https://github.com/ysiezar21/Tarea_corta_-1)