



# Análisis de Algoritmos

*UNAM, Facultad De Ciencias.*

**Profesora.** María de Luz Gasca Soto
**Ayudante.** Enrique Ehecatl Hernández Ferreiro
**Ayudante.** Brenda Margarita Becerra Ruíz

---

**Tarea 7: Ordenamientos II**

---

**Alumna:**
Villafán Flores María Fernanda (Ciencias de la Computación).

---

15 de noviembre de 2022
Ciudad de México.

1. El Problema de Selección consiste en encontrar el $k$-ésimo elemento más pequeño de un conjunto de $n$ datos, $k \leq n$.

   Proporcionar un algoritmo que solucione el Problema de Seleccin utilizando las estrategias de
   (a) Merge Sort; (b) Heap Sort. (c) Quick Sort; (d) Tree Sort.
   **Pregunta adicional:** ¿Se puede hacer sin ordenar toda la secuencia? Justificar

## Respuesta.

*a*) ***Merge Sort***

Primero, sabemos que este algoritmo se basa en la estrategia de *divide y vencerás*, esto es:

- *Divide:* Particiona el ejemplar original en ejemplares más pequeños, de manera recursiva, hasta reducirlo a ejemplares de tamaño uno.
- *Vence:* Resuelve para todos los ejemplares, iniciando con los de tamaño uno, continuando con los de tamaño dos, en cada paso va incrementando (duplicando) el tamaño del ejemplar hasta llegar al tamaño original.
- *Mezcla:* Combina las soluciones de los problemas, aplicadas a ejemplares pequeños para resolver problemas con ejemplares más grandes, de manera recursiva, hasta obtener la solución del problema original.

Entonces, veamos el algoritmo para encontrar al $k$-ésimo elemento más pequeño de un conjunto de $n$ datos (con $k \leq n$):

- ***Precondiciones:***
  - El conjunto de $n$ datos está contenido en una secuencia $S$ tal que es finita y se conforma de números enteros, donde $|S| = n$.
  - Los elementos de $S$ son comparables.
  - $k \leq n$.
- ***Algoritmo:***
  1) Dada la secuencia $S$, la vamos dividiendo por mitades hasta obtener subsecuencias de tamaño uno.
  2) Tomamos de dos en dos las subsecuencias de tamaño uno y las mezclamos de tal forma que estén ordenadas. Luego, tomamos las subsecuencias de tamaño dos que se generaron y volvemos a hacer este proceso de mezclarlas y ordenarlas hasta que tengamos una secuencia de tamaño $n$ ordenada.
  3) Ya que tenemos la secuencia $S$ ordenada, buscamos al índice $k$ en la secuencia y regresamos al elemento que se encuentra en dicha localidad de la secuencia ordenada.
- ***Postcondiciones:***

  El $k$-ésimo elemento más pequeño del conjunto de $n$ datos (con $k \leq n$).

**PREGUNTA ADICIONAL:**
Por la forma en la que *Merge Sort* ordena un conjunto de datos, NO se puede encontrar al $k$-ésimo elemento más pequeño hasta tener toda la secuencia ordenada. Esto debido a que primero divide la secuencia en subsecuencias más pequeñas que sean de tamaño uno, y conforme va mezclando y ordenando dichas subsecuencias tomadas de dos en dos, puede tener un candidato a ser el $k$-ésimo elemento más pequeño en una de las subsecuencias y cuando se tenga que mezclar la siguiente subsecuencia con la que tiene al candidato, puede ser que la otra subsecuencia tenga un elemento menor al candidato, y entonces este elemento será el $k$-ésimo elemento más pequeño. Este caso se puede dar en cada par de subsecuencias que se vayan mezclando, por lo que sólo se puede encontrar al $k$-ésimo elemento más pequeño de la secuencia original cuando esté ordenada.

*b*) ***Heap Sort***

Primero, sabemos que este algoritmo utiliza la estructura de datos denominada *Heaps Binarios*, por lo que requiere de las operaciones y acciones de estos como lo son:

- *Heapify:* Dado un árbol binario completo, se reorganizan los elementos del árbol para que cumplan con la relación de orden existente en el heap.
- *BorraMenor:* El nodo con la menor clave (o valor) se encuentra en la raíz, donde esto sucede para cada subárbol, entonces se van eliminando dichos elementos. Esta función requiere de:
  1) Extraer al elemento que está en la raíz del heap.
  2) Almacenar tal elemento en una lista.
  3) Eliminar la raíz del heap.
  4) Reorganizar el heap usando el proceso *Heapify*.
- Todo esto se lleva acabo de manera recursiva hasta que el heap queda vacío y por lo tanto, se tiene ordenada la secuencia en la lista.

Entonces, veamos el algoritmo para encontrar al $k$-ésimo elemento más pequeño de un conjunto de $n$ datos (con $k \leq n$):

- ***Precondiciones:***
  - El conjunto de $n$ datos está contenido en una secuencia $S$ tal que es finita y se conforma de números enteros, donde $|S| = n$.
  - Los elementos de $S$ son comparables.
  - $k \leq n$.
- ***Algoritmo:***
  1) Dada la secuencia $S$, vamos metiendo en un árbol binario cada elemento tal y como se encuentran en la secuencia.
  2) Empezando con el último subárbol, reestablecer heaps binarios hasta llegar a la raíz.
  3) Se reorganiza el heap tal que cada subárbol cumpla la propiedad de que el elemento con menor valor se encuentra en la raíz.

4) Extraer uno por uno elementos del heap, donde un contador lleva el número de elementos que se han sacado, y cuando el contador es igual a $k$, se regresa el último elemento que se extrajo del heap.

- ***Postcondiciones:***

  El $k$-ésimo elemento más pequeño del conjunto de $n$ datos (con $k \leq n$).

**PREGUNTA ADICIONAL:**
Como *Heap Sort* utiliza la estructura de datos *Heaps Binarios*, cuando ya se tiene construido, sí se puede encontrar al $k$-ésimo elemento más pequeño sin necesidad de terminar de vaciar el heap (lo que significaría que ya tenemos la secuencia $S$ almacenada en una lista ordenada) porque únicamente debemos extraer elementos hasta tener al $k$-ésimo; y ya que utilizamos un *MinHeap*, podemos garantizar que el $k$-ésimo elemento que se extrajo del heap es el $k$-ésimo elemento más pequeño del conjunto.

*c*) ***Quick Sort***

Primero, sabemos que este algoritmo se basa en la estrategia de *divide y vencerás*, esto es:

- *Divide:* La secuencia es dividida (particionada) en dos subsecuencias. Todos los elementos de la primera subsecuencia son menores o iguales a los elementos de la segunda, además, ambas subsecuencias no pueden ser vacías al mismo tiempo.
- *Vence:* Las dos subsecuencias son ordenadas mediante llamadas recursivas a *Quick Sort*.

Entonces, veamos el algoritmo para encontrar al $k$-ésimo elemento más pequeño de un conjunto de $n$ datos (con $k \leq n$):

- ***Precondiciones:***
  - El conjunto de $n$ datos está contenido en una secuencia $S$ tal que es finita y se conforma de números enteros, donde $|S| = n$.
  - Los elementos de $S$ son comparables.
  - $k \leq n$.
- ***Algoritmo:***
  1) Seleccionamos al primer elemento de la secuencia $S$ como *pivote*.
  2) Inicializamos dos índices auxiliares $i, j$ tal que $i$ apunta al primer elemento de la secuencia y $j$ apunta al último elemento de la secuencia.
  3) Vamos recorriendo $i$ a la derecha hasta que encuentre un elemento que sea mayor al *pivote*.
  4) De forma similar al punto anterior, vamos recorriendo $j$ a la izquierda hasta que encuentre un elemento que sea menor o igual al *pivote*.
  5) Hacemos "swap" de los elementos encontrados en el punto 3) y 4).
  6) Repetimos los puntos 3), 4) y 5) hasta que $i, j$ se "crucen", es decir, que $i$ sea mayor o igual a $j$.

7) Hacemos "swap" del *pivote* con el elemento que está en la posición del índice $j$.

8) Preguntamos lo siguiente para $k$:

   ◦ Si $k$ es igual al índice $j$, regresamos al elemento que está en la posición del índice $j$.

   ◦ Si $k$ es menor al índice $j$, repetimos los puntos 1) a 7) con la subsecuencia que está a la izquierda del *pivote*.

   ◦ Si $k$ es mayor al índice $j$, repetimos los puntos 1) a 7) con la subsecuencia que está a la derecha del *pivote*.

- ***Postcondiciones:***

  El $k$-ésimo elemento más pequeño del conjunto de $n$ datos (con $k \leq n$).

**PREGUNTA ADICIONAL:**
Como *Quick Sort* acomoda al elemento que tomamos como *pivote* en su posición correcta en la secuencia, sí podemos encontrar al $k$-ésimo elemento más pequeño sin necesidad de terminar de ordenar toda la secuencia. Esto debido a que al comparar $k$ con el índice $j$ (que fue el índice donde se acomodó al *pivote* en su posición correcta), ya sea que regresamos el índice $j$, o realizamos *Quick Sort* sobre una de las dos subsecuencias (izquierda o derecha) que tenemos, por lo que descartamos a todos los elementos de la subsecuencia que ya no vamos a ordenar.

*d*) ***Tree Sort***

Primero, sabemos que este algoritmo utiliza la estructura de datos denominada *Árbol Binario de Búsqueda* como estructura auxiliar para ordenar los datos de una secuencia dada. Es decir, lo único que hace este proceso es guardar la secuencia $S$ dada en un árbol de búsqueda binaria, con las reglas de éste, para después hacer un recorrido en-orden (*in-order*) sobre el árbol, almacenando en una lista los datos recuperados del árbol.

Entonces, veamos el algoritmo para encontrar al $k$-ésimo elemento más pequeño de un conjunto de $n$ datos (con $k \leq n$):

- ***Precondiciones:***
  - El conjunto de $n$ datos está contenido en una secuencia $S$ tal que es finita y se conforma de números enteros, donde $|S| = n$.
  - Los elementos de $S$ son comparables.
  - $k \leq n$.
- ***Algoritmo:***

  1) Dada la secuencia $S$, vamos metiendo en un árbol binario cada elemento tal y como se encuentran en la secuencia, de tal forma que se cumpla la siguiente propiedad:

     Para cada subárbol, el elemento que está a la izquierda es menor al elemento que está en la raíz y el elemento que está a la derecha es mayor o igual al elemento que está en la raíz.

2) Recorremos el árbol en recorrido *in-orden* hasta que el último elemento por el que pasamos tenga índice igual a $k$ y regresamos al elemento.

- ***Postcondiciones:***

  El $k$-ésimo elemento más pequeño del conjunto de $n$ datos (con $k \leq n$).

**PREGUNTA ADICIONAL:**
Como *Tree Sort* utiliza la estructura de datos *Árbol Binario*, implícitamente ya tenemos la secuencia ordenada porque sólo tenemos que realizar el recorrido *in-orden* para obtener al $k$-ésimo elemento más pequeño. Entonces, esto significa que NO podemos encontrar al $k$-ésimo elemento más pequeño hasta ordenar toda la secuencia porque la secuencia ya está ordenada en el árbol.

■

2. **Afirmación.** El sub-algoritmo Partition del Algoritmo Quick Sort puede ser usado para encontrar la mediana de una lista de valores, de tamaño $n$, en tiempo $O(n)$.
   **a)** Mostrar que la afirmación es verdadera, diseñando y presentando el algoritmo correspondiente
   **b)** Justificar detalladamente que se alcanza el tiempo dado.

## Respuesta.

*a*) Primero, recordemos que la *mediana* de una lista de valores es el valor que ocupa el lugar central de todos los datos cuando éstos están ordenados de menor a mayor.

Como en el *Ejercicio 1 (c)* vimos un algoritmo para encontrar al $k$-ésimo elemento más pequeño de un conjunto de $n$ datos (con $k \leq n$) utilizando la estrategia de *Quick Sort*, entonces podemos retomar este algoritmo y modificarlo para encontrar una $k$ específica:

$$k = \left\lceil \left(minIndice + maxIndice\right) \text{ div } 2 \right\rceil$$

Entonces, veamos el algoritmo modificado para encontrar al $k$-ésimo elemento de un conjunto de $n$ datos (con $k \leq n$):

- ***Precondiciones:***
  - El conjunto de $n$ datos está contenido en una secuencia $S$ tal que es finita y se conforma de números enteros, donde $|S| = n$.
  - Los elementos de $S$ son comparables.
  - $k \leq n$.
- ***Algoritmo:***

1) Utilizamos el menor y el mayor índice de la secuencia $S$ para calcular el valor de $k$ como:

$$k = \left\lceil \left(minIndice + maxIndice\right) \text{ div } 2 \right\rceil$$

2) Seleccionamos al primer elemento de la secuencia $S$ como *pivote*.

3) Inicializamos dos índices auxiliares $i, j$ tal que $i$ apunta al primer elemento de la secuencia y $j$ apunta al último elemento de la secuencia.

4) Vamos recorriendo $i$ a la derecha hasta que encuentre un elemento que sea mayor al *pivote*.

5) De forma similar al punto anterior, vamos recorriendo $j$ a la izquierda hasta que encuentre un elemento que sea menor o igual al *pivote*.

6) Hacemos "swap" de los elementos encontrados en el punto 4) y 5).

7) Repetimos los puntos 4), 5) y 6) hasta que $i, j$ se "crucen", es decir, que $i$ sea mayor o igual a $j$.

8) Hacemos "swap" del *pivote* con el elemento que está en la posición del índice $j$.

9) Preguntamos lo siguiente para $k$:

- Si $k$ es igual al índice $j$, regresamos al elemento que está en la posición del índice $j$.
- Si $k$ es menor al índice $j$, repetimos los puntos 2) a 8) con la subsecuencia que está a la izquierda del *pivote*.
- Si $k$ es mayor al índice $j$, repetimos los puntos 2) a 8) con la subsecuencia que está a la derecha del *pivote*.

- ***Postcondiciones:***

  El $k$-ésimo elemento del conjunto de $n$ datos (con $k \leq n$).

*b*) Para el análisis de complejidad total del algoritmo, veamos lo siguiente:

1) Determinar el mínimo y el máximo índice de una secuencia toma tiempo $O(1)$, a su vez, calcular el valor de $k$ utilizando estos valores obtenidos.

2) Determinar un elemento de la secuencia $S$ como *pivote* toma tiempo $O(1)$.

3) Crear dos índices auxiliares $i, j$ toma tiempo $O(1)$.

4) Recorrer los índices auxiliares $i, j$ hacia la izquierda y hacia la derecha respectivamente por la secuencia $S$ toma tiempo $O(n)$.

5) Hacer "swap" de los elementos a los que apuntan los índices $i, j$ toma tiempo $O(1)$.

6) Realizar el punto 5) para todos los elementos de la secuencia $S$ toma tiempo $O(n)$.

7) A diferencia de la versión original de *Quick Sort*, la cuál toma cada subsecuencia (izquierda ***y*** derecha) generadas con respecto al *pivote* para realizar el mismo proceso para cada una de ellas, éste algoritmo modificado que dimos únicamente se queda con una de las dos subsecuencias (izquierda ***o*** derecha) generadas con respecto al *pivote* y realiza

el mismo proceso nada más para **una** de las subsecuencias, donde esto sucede por cada llamada recursiva que se haga al mismo proceso. Entonces, esto reduce la complejidad de $O(n \cdot \log(n))$ a $O(n)$.

Por lo tanto, podemos concluir que este algoritmo modificado tiene una complejidad total de $O(n)$.

■

3. Sea $\mathcal{L}$ una secuencia de $n$ números enteros diferentes. Suponga que los elementos $x$ de $\mathcal{L}$ están en el intervalo $[1, 2000]$.
   **a) Diseñar** un algoritmo de orden lineal que ordene los elementos de la secuencia $\mathcal{L}$.
   **b)** Justificar detalladamente que se alcanza el tiempo solicitado.
   **c)** El algoritmo resultante, ¿viola la cota mínima de ordenamiento? ¿ Por qué? Justifica con detalle tu respuesta.

## Respuesta.

*a*) Primero, para resolver este ejercicio, nos basaremos en la estrategia del algoritmo *Counting Sort* que consiste en determinar para cada elemento $x$ en la secuencia $\mathcal{L}$, su número de apariciones en ésta y, por lo tanto, el número de elementos menores a él. Ésta información podrá ser usada para posicionar cada elemento directamente en su posición correspondiente en la secuencia $\mathcal{L}$ ordenada.

Entonces, veamos el algoritmo para ordenar los elementos de la secuencia $\mathcal{L}$:

- ***Precondiciones:***
  - Los elementos de $\mathcal{L}$ son números enteros no negativos y comparables.
  - Conocer el rango entre los que se encuentran los elementos de $\mathcal{L}$.
- ***Algoritmo:***
  1) Dado el rango, determinamos el mínimo elemento y el máximo elemento de la secuencia $\mathcal{L}$.
  2) Creamos un vector auxiliar para contar el número de apariciones de cada elemento de la secuencia, donde este vector es de tamaño $k$ (con $k$ = máximo elemento).
  3) Estimamos la cantidad de elementos menores al elemento en revisión al ir sumando los valores contiguos a la casilla en revisión.
  4) Creamos un arreglo de tamaño $n$ que contendrá a los elementos ya ordenados.
  5) Recorremos la secuencia y el arreglo auxiliar colocando al elemento examinado en su posición correspondiente en el arreglo final.

- ***Postcondiciones:***

  La secuencia $\mathcal{L}$ de tamaño $n$ ordenada de forma ascendente.

*b*) Para el análisis de complejidad total del algoritmo, veamos lo siguiente:

1) Determinar el mínimo y el máximo elemento de una secuencia toma tiempo $O(1)$.

2) Crear un vector auxiliar el cuál tiene tamaño $k$ toma tiempo $O(1)$.

3) Contar el número de apariciones de cada elemento en la secuencia toma tiempo $O(n)$.

4) Estimar la cantidad de elementos menores al elemento en revisión y realizar las sumas pertinentes toma tiempo $O(k)$, ya que se emplean $(k-1)$ sumas y suponemos que cada suma toma tiempo $O(1)$.

5) Crear un arreglo de tamaño $n$ que contendrá a los elementos ya ordenados toma tiempo $O(1)$.

6) Recorrer tanto la secuencia como el arreglo auxiliar e insertar un valor en el arreglo toma tiempo $O(n+k+n) = O(2n+k) = O(n)$.

De lo anterior, tenemos:

$$O(1) + O(1) + O(n) + O(k) + O(1) + O(n) = O(n)$$

Por lo tanto, podemos concluir que este algoritmo tiene una complejidad total de $O(n)$.

*c*) Recordemos que la ***Cota Mínima de Ordenamiento*** indica que el mejor desempeño computacional que se puede alcanzar (en el peor de los casos) para las técnicas basadas en comparaciones es de $n \cdot \log(n)$.

Ahora, como el algoritmo propuesto en el inciso *a*) NO utiliza comparaciones ya que se basa en la estrategia del algoritmo *Counting Sort*, tenemos que NO viola la cota mínima de ordenamiento.

■

4. Considerar dos versiones de Quick Sort, aplicado a una secuencia $A$ de datos $n$ datos; $A[l..r]$
QuickSort_1 que toma como pivote al elemento $A[n \text{ div } 2]$;
QuickSort_2 que toma como pivote al elemento que resulta ser la mediana de $\{A[l],\ A[(l+r) \text{ div } 2],\ A[r]\}$.
**(a)** Dar un ejemplo de una lista de al menos 35 valores donde el desempeño computacional de QuickSort_2 sea mejor que el de QuickSort_1
**(b)** Dar un ejemplo de una lista de al menos 35 valores donde el desempeño computacional de QuickSort_1 sea mejor que el de QuickSort_2
**(c)** Presentar la traza de ambas ejecuciones.

**Sin respuesta.**

■

5. Opcional. Consideremos $k$ secuencias de elementos $S_1, S_2, ..., S_k$. Cada secuencia $S_j$ está ordenada. Además, $|S_1| + |S_2| + ... + |S_k| = n$, el número total de elementos.
   **(a) Diseñar** un algoritmo que genere una secuencia ordenada de estos $n$ elementos. Su algoritmo debe tener complejidad $O(n \log k)$.
   **(b) Verificar** que el algoritmo propuesto alcanza la cota deseada.

**Sin respuesta.**

■