# MergeSort

El algoritmo MERGESORT funciona en dos fases principales:

1. **División recursiva**: El array se divide repetidamente en subarrays hasta que cada subarray tiene un solo elemento.
2. **Mezcla (merge)**: Luego, se combinan los subarrays de manera ordenada.

**Complejidad del Algoritmo:**

1. **División recursiva**:
   * En cada nivel de recursión, el array se divide a la mitad. Esto genera un árbol de recursión cuya altura es logarítmica, es decir,
   * Cada nivel del árbol de recursión procesa elementos en total (considerando todas las divisiones en ese nivel).
2. **Mezcla (merge)**:
   * El proceso de mezcla compara y combina elementos de subarrays. En cada nivel de recursión, la operación de mezcla toma tiempo proporcional a porque todos los elementos deben ser procesados.
   * Esto se repite a lo largo de los niveles.

**Complejidades:**

* **O(n log n)**: MERGESORT tiene una complejidad temporal en el peor caso .  
  Esto se debe a que el proceso de dividir el array toma niveles, y en cada nivel se realizan operaciones para la mezcla. Así que, en total, el peor caso es .
* **Ω(n log n)**: MERGESORT también tiene una complejidad temporal en el mejor caso . Incluso si el array ya está ordenado, el algoritmo realiza las mismas divisiones y combinaciones. No hay un atajo que lo haga más rápido, por lo que el mejor caso también es .
* **Ɵ(n log n)**: MERGESORT es un algoritmo con complejidad, lo que significa que tanto el mejor como el peor caso tienen la misma complejidad temporal. Esto indica que la función de tiempo está limitada de manera asintótica tanto superior como inferiormente por .

**Max Heapify**

Para el algoritmo Max Heapify se busca encontrar una función que relacione la cantidad de pasos con su complejidad, para ello se tiene en cuenta el código adjunto, resumiendo la búsqueda de la función de complejidad de la siguiente manera:

**Función heapify**:

**Complejidad del peor caso**: La función **heapify** se llama sobre un nodo y potencialmente realiza comparaciones y swaps a lo largo de un camino en el árbol.

Para un nodo en un árbol de altura h, la complejidad en el peor caso es O(h), ya que puede haber hasta h niveles en los que se realiza una comparación o swap.

En un **heap** de n elementos, la altura máxima h es O (log n).

Para las diversas iteraciones del ciclo for en la función **head\_sort**, se ejecuta veces (empezando desde , y en cada iteración se llama a ***heapity***

Como ***heapify*** tiene una complejidad O (log n), la complejidad total para construir el **Max-heap** es, y se puede modelar una aproximación a la siguiente sumatoria:

En consecuencia, la función que mejor describiría la complejidad del algoritmo es:

**Quicksort**

En la implementación del algoritmo de **QuickSort se realiza un análisis de los ciclos de la siguiente manera:**

Para el buble **for** Este bucle se ejecuta desde low hasta high - 1, lo que implica que se ejecuta (O(n)) veces, donde (n) es el tamaño del subarray.

Para las operaciones **dentro del bucle**, Cada iteración del bucle realiza comparaciones y, potencialmente, intercambios, ambos de los cuales son operaciones de tiempo constante (O (1)).

Por lo tanto, la complejidad de la función **partition** es (O(n)).

La función ***quickSor***t se llama recursivamente dos veces para cada partición. En el mejor caso, cada partición divide el array en dos mitades iguales, lo que lleva a una profundidad de recursión de (n log n).

Por lo tanto, el análisis de complejidad para este algoritmo en el peor de los casos es

Y para el mejor de los casos debido al numero de iteraciones es: