ALGORITMOS LOGARITMICOS

CHAVEZ MERINO RICARDO

ESTRUCTURA DE DATOS

SOFT02\_03

INGENIERIA EN SOFTWARE

INTRODUCCION

Primero que nada, definamos que es son estos métodos de búsqueda dentro del tema de los algoritmos de ordenamiento y porque este es muy necesario en el campo de la programación y claro como la ingeniería en software no siendo solo programador debe saberlo.

Bueno podemos decir que un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden esto englobando de una manera básica y superficial el concepto. Estos tienen una función y nos dice que los ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, BubbleSort fue analizado desde [1956](https://www.ecured.cu/1956). Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy y por eso la importancia de los mismos.

Este resumen consta con tres métodos de ordenamiento usados comúnmente, uno mas que otro, pero esto de debe a sus propiedades de cada uno como a su estabilidad o no.

Se describe el método quicksort, heapsort y mergesort y sus respectivos procedimientos, todo a nivel teórico y con ejemplos simples sin ir aun a la programación o declaración de estas, pues recordemos que este es un resumen recopilatorio si se puede decir así.

DESARROLLO

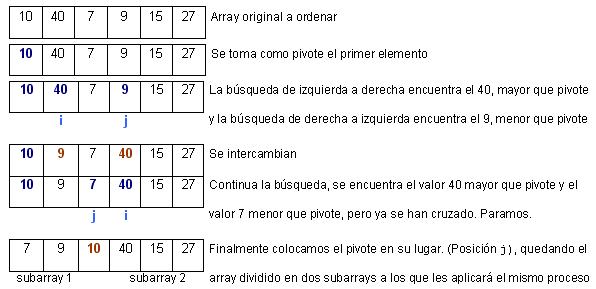
**Método de Ordenación Rápida (Quicksort)**

El método de ordenación rápida (quicksort) para ordenar o clasificar un vector o lista de elementos (array) se basa en el hecho de que es más rápido y fácil de ordenar dos listas pequeñas que una lista grande. Se denomina método de ordenación rápida porque, en general, puede ordenar una lista de datos mucho más rápidamente que cualquiera de los métodos de ordenación ya estudiados.

El método se basa en la estrategia típica de “divide y vencerás”. La lista a clasificar almacenada en un vector o array se divide (parte) en dos sublistas: una con todos los valores menores o iguales a un cierto valor específico y otra con todos los valores mayores que ese valor.

El método consta de los siguientes pasos:

1. Elegir un elemento de la lista de elementos a ordenar, al que llamaremos [pivote](https://www.ecured.cu/index.php?title=Pivote&action=edit&redlink=1).
2. Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
3. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
4. Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.



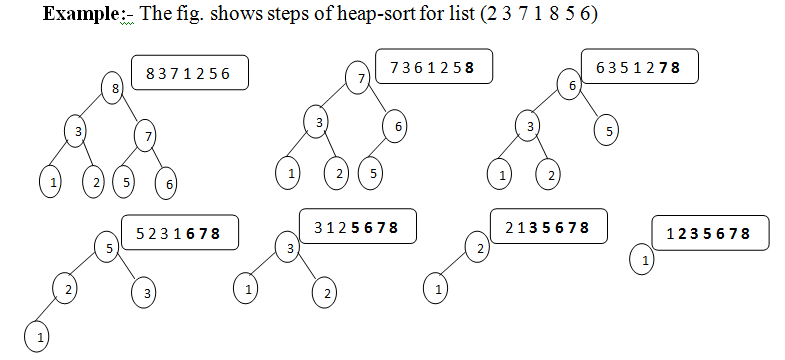
Ejemplo QuickSort

**Método de Ordenación por Montículos (Heapsort)**

Es un algoritmo de ordenación no [recursivo](https://www.ecured.cu/Recursividad), no estable. Este [algoritmo](https://www.ecured.cu/Algoritmo) consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

Este método costa de los siguientes pasos:

1. Construir un Heap sobre el arreglo a ordenar (BUILD\_HEAP), en orden contrario al orden de ordenación. Por ejemplo, para ordenar ascendentemente un arreglo se construye un HEAP sobre él de forma descendiente (el mayor elemento queda en la raíz)
2. Intercambiar la raíz del Heap, posición 0, con la última posición en el Heap.
3. Disminuir el tamaño del Heap.
4. Reconstruir el Heap aplicando Heapify en la primera posición.
5. Repetir los pasos 2, 3 y 4 mientras el tamaño del Heap sea mayor que uno.



Ejemplo HeapSort

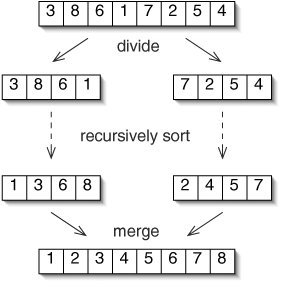
**Método de Ordenación por Mezclas Sucesivas (Mergesort)**

Se aplica la [técnica](https://www.ecured.cu/T%C3%A9cnica) divide y vencerás, dividiendo la [secuencia](https://www.ecured.cu/index.php?title=Secuencia&action=edit&redlink=1) de datos en dos subsecuencias hasta que las subsecuencias tengan un único [elemento](https://www.ecured.cu/index.php?title=Elemento&action=edit&redlink=1), luego se ordenan mezclando dos subsecuencias ordenadas en una [secuencia ordenada](https://www.ecured.cu/index.php?title=Secuencia_ordenada&action=edit&redlink=1), en forma sucesiva hasta obtener una [secuencia](https://www.ecured.cu/index.php?title=Secuencia&action=edit&redlink=1) única ya ordenada. Si n = 1 solo hay un [elemento](https://www.ecured.cu/index.php?title=Elemento&action=edit&redlink=1) por ordenar, sino se hace una [ordenación](https://www.ecured.cu/index.php?title=Ordenaci%C3%B3n&action=edit&redlink=1) de [mezcla](https://www.ecured.cu/index.php?title=Mezcla&action=edit&redlink=1) de la primera mitad del [arreglo](https://www.ecured.cu/index.php?title=Arreglo&action=edit&redlink=1) con la segunda mitad. Las dos mitades se ordenan de igual forma.

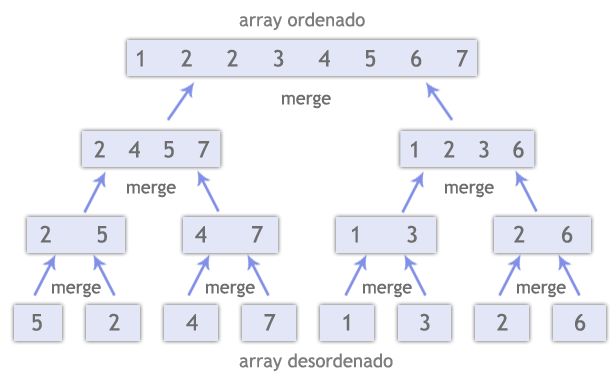
MergeSort es un ordenamiento estable, paraleliza mejor, y es más eficiente manejando medios secuenciales de [acceso](https://www.ecured.cu/index.php?title=Acceso&action=edit&redlink=1) lento. MergeSort es a menudo la mejor opción para ordenar una [lista enlazada](https://www.ecured.cu/index.php?title=Lista_enlazada&action=edit&redlink=1): en esta situación es relativamente fácil implementar MergeSort de manera que sólo requiera O(1) espacio extra, y el mal rendimiento de las listas enlazadas ante el [acceso aleatorio](https://www.ecured.cu/index.php?title=Acceso_aleatori&action=edit&redlink=1) hace que otros algoritmos como [quicksort](https://www.ecured.cu/index.php?title=Quicksort&action=edit&redlink=1) den un bajo [rendimiento](https://www.ecured.cu/index.php?title=Rendimiento&action=edit&redlink=1), y para otros como [heapsort](https://www.ecured.cu/Heapsort" \o "Heapsort) sea algo imposible. El algoritmo de MergeSort es un [algortimo](https://www.ecured.cu/index.php?title=Algortimo&action=edit&redlink=1" \o "Algortimo (la página no existe)) de sort que presenta algunas propiedades interesantes.

Este método consta de los siguientes pasos:

1. Ordenar una secuencia S de elementos:
   1. Si S tiene uno o ningún elemento, está ordenada
   2. Si S tiene al menos dos elementos se divide en dos secuencias S1 y S2, S1 conteniendo los primeros n/2, y S2 los restantes.
   3. Ordenar S1 y S2, aplicando recursivamente este procedimiento.
   4. Mezclar S1 y S2 ordenadamente en S
2. Mezclar dos secuencias ordenadas S1 y S2 en S:
   1. Se tienen referencias al principio de cada una de las secuencias a mezclar (S1 y S2).
   2. Mientras en alguna secuencia queden elementos, se inserta en la secuencia resultante (S) el menor de los elementos referenciados y se avanza esa referencia una posición.



Ejemplo 1. MergeSort



Ejemplo 2 MergeSort

Causas de Complejidad

**Órdenes de Complejidad**

* La familia O(*f(n)*) define un *Orden de Complejidad*. Elegiremos como representante de este *Orden de Complejidad* a la función *f(n)* más sencilla perteneciente a esta familia.
* Las funciones de complejidad algorítmica más habituales en las cuales el único factor del que dependen es el tamaño de la [muestra](https://www.monografias.com/trabajos11/tebas/tebas.shtml) de entrada *n*, ordenadas de mayor a menor eficiencia son:

|  |  |
| --- | --- |
| O(1) | Orden constante |
| O(*log n*) | Orden logarítmico |
| O(*n*) | Orden lineal |
| O(*n log n*) | Orden cuasi-lineal |
| O(*n2*) | Orden cuadrático |
| O(*n3*) | Orden cúbico |
| O(*na*) | Orden polinómico |
| O(2*n*) | Orden exponencial |
| O(*n!*) | Orden factorial |

Se identifica una Jerarquía de Ordenes de Complejidad que coincide con el orden de la tabla mostrada; jerarquía en el sentido de que cada orden de complejidad inferior tiene a las superiores como subconjuntos.

* + O(1): Complejidad constante. Cuando las instrucciones se ejecutan una vez.
  + O(*log n*): Complejidad logarítmica. Esta suele aparecer en determinados algoritmos con iteración o recursión no estructural, ejemplo la búsqueda binaria.
  + O(*n*): Complejidad lineal. Es una complejidad buena y también muy usual. Aparece en la [evaluación](https://www.monografias.com/trabajos11/conce/conce.shtml) de bucles simples siempre que la complejidad de las instrucciones interiores sea constante.
  + O(*n log n*): Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación quicksort y se considera una buena complejidad. Si *n* se duplica, el tiempo de ejecución es ligeramente mayor del doble.
  + O(*n2*): Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si n se duplica, el tiempo de ejecución aumenta cuatro veces.
  + O(*n3*): Complejidad cúbica. Suele darse en bucles con triple anidación. Si *n* se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de n empieza a crecer dramáticamente.
  + O(*na*): Complejidad polinómica (*a* > 3). Si a crece, la complejidad del programa es bastante mala.
  + O(*2n*): Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas. N

Diferencias entre los Métodos de Ordenación

Quicksort

Tiene aparentemente la propiedad

de trabajar mejor para elementos de entrada desordenados completamente, que para

elementos semi-ordenados. Esta situación es precisamente la opuesta al ordenamiento

de burbuja. Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es

más rápido y fácil ordenar dos arreglos o listas de datos pequeños , que un arreglo o

lista grande. Normalmente al inicio de la ordenación se escoge un elemento

aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a

que el arreglo está ordenado respecto al punto de división o la mitad del arreglo. Se

podrá garantizar que los elementos a la izquierda de la mitad son los menores y los

elementos a la derecha son los mayores.

Tiene aparentemente la propiedad

de trabajar mejor para elementos de entrada desordenados completamente, que para

elementos semi-ordenados. Esta situación es precisamente la opuesta al ordenamiento

de burbuja. Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es

más rápido y fácil ordenar dos arreglos o listas de datos pequeños , que un arreglo o

lista grande. Normalmente al inicio de la ordenación se escoge un elemento

aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a

que el arreglo está ordenado respecto al punto de división o la mitad del arreglo. Se

podrá garantizar que los elementos a la izquierda de la mitad son los menores y los

elementos a la derecha son los mayores.

Tiene aparentemente la propiedad

de trabajar mejor para elementos de entrada desordenados completamente, que para

elementos semi-ordenados. Esta situación es precisamente la opuesta al ordenamiento

de burbuja. Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es

más rápido y fácil ordenar dos arreglos o listas de datos pequeños , que un arreglo o

lista grande. Normalmente al inicio de la ordenación se escoge un elemento

aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a

que el arreglo está ordenado respecto al punto de división o la mitad del arreglo. Se

podrá garantizar que los elementos a la izquierda de la mitad son los menores y los

elementos a la derecha son los mayores.

Tiene aparentemente la propiedad

de trabajar mejor para elementos de entrada desordenados completamente, que para

elementos semi-ordenados. Esta situación es precisamente la opuesta al ordenamiento

de burbuja. Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es

más rápido y fácil ordenar dos arreglos o listas de datos pequeños , que un arreglo o

lista grande. Normalmente al inicio de la ordenación se escoge un elemento

aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a

que el arreglo está ordenado respecto al punto de división o la mitad del arreglo. Se

podrá garantizar que los elementos a la izquierda de la mitad son los menores y los

elementos a la derecha son los mayores.

Tiene aparentemente la propiedad de trabajar mejor para elementos de entrada desordenados completamente, que para elementos semi-ordenados. Esta situación es precisamente la opuesta al ordenamientode burbuja. Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es más rápido y fácil ordenar dos arreglos o listas de datos pequeños , que un arreglo o lista grande. Normalmente al inicio de la ordenación se escoge un elemento aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a que el arreglo está ordenado respecto al punto de división o la mitad del arreglo. Se podrá garantizar que los elementos a la izquierda de la mitad son los menores y los elementos a la derecha son los mayores.

Eficiencia: En tiempo es nlogn y en espacio es N

Datos: Se puede hacer para una gran lista de datos, ya que es eficiente para ello.

Ventajas:

* Requiere de pocos recursos en comparación a otros métodos de ordenamiento.
* En la mayoría de los casos, se requiere aproximadamente N log N operaciones.
* Ciclo interno es extremadamente corto.
* No se requiere de espacio adicional durante ejecución (in-place processing).

Desventajas:

* Se complica la implementación si la recursión no es posible.
* Peor caso, se requiere N2
* Un simple error en la implementación puede pasar sin detección, lo que provocaría un rendimiento pésimo.
* No es útil para aplicaciones de entrada dinámica, donde se requiere reordenar una lista de elementos con nuevos valores.
* Se pierde el orden relativo de elementos idénticos.

Aplicaciones:

* Para ordenar una lista de números/nombres.
* Utilización antes de implementar una búsqueda binaria.
* Utilizado como el método de ordenamiento en tarjetas gráficas.

Heapsort

Ventajas:

* Su desempeño es tan bueno como el quicksort, en lo peores casos se comporta mejor que este.
* No utiliza memoria adicional.

Desventajas:

* No es estable, ya que se comporta de manera ineficaz don datos del mismo valor
* Es un método mas complejo

1. Fue el primer algoritmo que surgió de los que ordenan en O(n log n).
2. Ordena en el lugar
   1. -No utiliza memoria auxiliar
3. No es Estable

Mergesort

Se aplica la [técnica](https://www.ecured.cu/T%C3%A9cnica) divide y vencerás, dividiendo la [secuencia](https://www.ecured.cu/index.php?title=Secuencia&action=edit&redlink=1) de datos en dos subsecuencias hasta que las subsecuencias tengan un único [elemento](https://www.ecured.cu/index.php?title=Elemento&action=edit&redlink=1), luego se ordenan mezclando dos subsecuencias ordenadas en una [secuencia ordenada](https://www.ecured.cu/index.php?title=Secuencia_ordenada&action=edit&redlink=1), en forma sucesiva hasta obtener una [secuencia](https://www.ecured.cu/index.php?title=Secuencia&action=edit&redlink=1) única ya ordenada.

 es más eficiente manejando medios secuenciales de [acceso](https://www.ecured.cu/index.php?title=Acceso&action=edit&redlink=1) lento.

MergeSort es a menudo la mejor opción para ordenar una [lista enlazada](https://www.ecured.cu/index.php?title=Lista_enlazada&action=edit&redlink=1): en esta situación es relativamente fácil implementar MergeSort de manera que sólo requiera O(1) espacio extra, y el mal rendimiento de las listas enlazadas ante el [acceso aleatorio](https://www.ecured.cu/index.php?title=Acceso_aleatori&action=edit&redlink=1) hace que otros algoritmos como quicksort den un bajo [rendimiento](https://www.ecured.cu/index.php?title=Rendimiento&action=edit&redlink=1), y para otros como [heapsort](https://www.ecured.cu/Heapsort" \o "Heapsort) sea algo imposible.

* Es Estable.
* Memoria Auxiliar O(n).
* No ordena en el lugar.
* Es O(n log n).

CONCLUSION

A un nivel teórico este tipo de temas son entendibles pues siguen reglas. Por lo cual se me hace mas digeribles y puedo entenderlos. Bueno como llevo meses programando espero que hacer este resumen y buscar información de cada método, así como sus características y limitantes de cada uno me ayude a poder usarlos.

Yo ya conocía el método de ordenamiento quicksort, pero apenas conocí los otros dos como es el heapsort y el mergesort y bueno también se que aun hay mas métodos. Pero estos apenas ya entiendo un poco más su funcionamiento.

El uso de estos métodos sin duda se nos hará difícil las primeras veces que los implementemos, pero a largo plazo nos serán muy útiles y al final estaremos agradecidos.

BIBLIOGRAFIA

<https://combomix.net/wp-content/uploads/2017/03/Fundamentos-de-programaci%C3%B3n-4ta-Edici%C3%B3n-Luis-Joyanes-Aguilar-2.pdf>

<https://www.ecured.cu/MergeSort#Propiedades>

<https://www.studocu.com/es-mx/document/cetys-universidad/estructura-de-datos/ejercicios-obligatorios/metodos-ordenamiento/3115637/view>

<https://www.ecured.cu/QuickSort>

<https://elbauldelprogramador.com/algoritmos-de-ordenacion/>