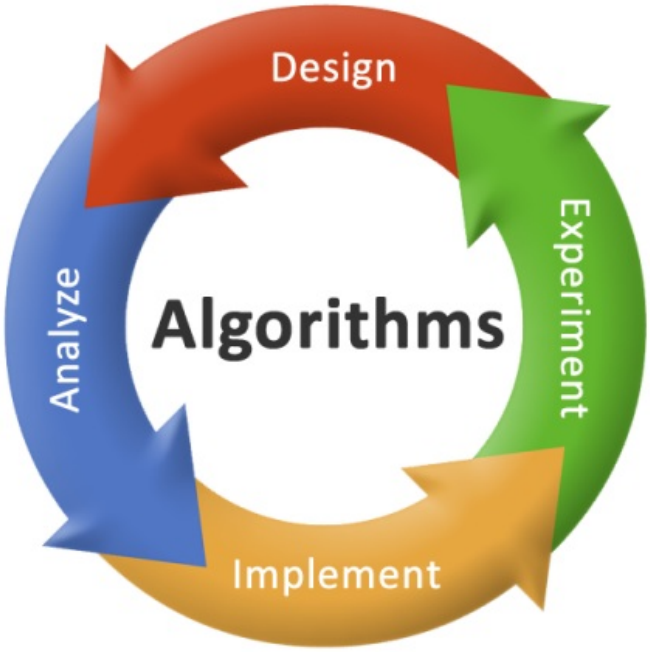
Ordenación por inserción y Merge Sort.

Comparativa de algoritmos de ordenación.

Joaquín Sanchíz Navarro y Sergio García de la Iglesia



Algoritmos de ordenación

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, es decir, el resultado de salida ha de ser un reordenamiento de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. 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.

Clasificación

Los algoritmos de ordenamiento se pueden clasificar en las siguientes maneras:

- Según el lugar donde se realice la ordenación
    - Algoritmos de ordenamiento interno: en la memoria del ordenador.
    - Algoritmos de ordenamiento externo: en un lugar externo como un disco duro.
- Por el tiempo que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:
    - Algoritmos de ordenación natural: Tarda lo mínimo posible cuando la entrada está ordenada.
    - Algoritmos de ordenación no natural: Tarda lo mínimo posible cuando la entrada está inversamente ordenada.
- Por estabilidad: un ordenamiento estable mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Por ejemplo, si una lista ordenada por fecha se reordena en orden alfabético con un algoritmo estable, todos los elementos cuya clave alfabética sea la misma quedarán en orden de fecha.

Características

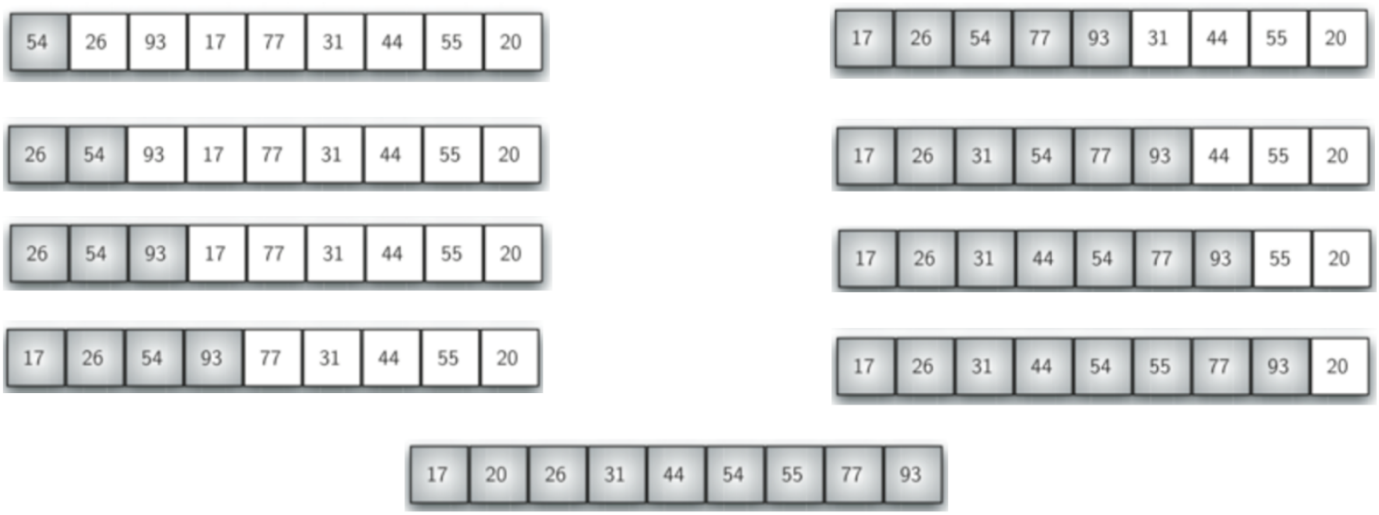
- Complejidad computacional (peor caso, caso promedio y mejor caso) en términos de n, el tamaño de la lista o arreglo. Para esto se usa el concepto de *orden* de una función y se usa la notación O(n). El mejor comportamiento para ordenar (si no se aprovecha la estructura de las claves) es O(n log n). Los algoritmos más simples son cuadráticos, es decir O(n^2).
- Uso de memoria y otros recursos computacionales. También se usa la notación O(n).

Ordenación por inserción (Insertion Sort)

El **ordenamiento por inserción** (**insertion sort**) es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria. Requiere **O(n²)** operaciones para ordenar una lista de **n** elementos. (En el peor de los casos)
Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay ***k*** elementos ordenados de menor a mayor, se toma el elemento ***k+1*** y se compara

con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se *inserta* el elemento ***k+1*** debiendo desplazarse los demás elementos.

```
function insertionSort(array A)
    for i from 1 to length[A]-1 do
        value := A[i]
        j := i-1
        while j >= 0 and A[j] > value do
            A[j+1] := A[j]
            j := j-1
        done
        A[j+1] = value
    done
```



Análisis de la complejidad del Insertion Sort

Tomemos una situación en donde llamamos a insert y el valor que se inserta en una sublista es menor que cualquier elemento en el sublista. Por ejemplo, si estamos insertando 0 en el sublista [2, 3, 5, 7, 11], entonces todos los elementos del sublista tienen que recorrerse una posición hacia la derecha. Así que, en general, si estamos insertando en un sublista con k elementos, todos los k elementos podrían tener que recorrerse una posición. En lugar de contar exactamente cuántas líneas de código necesitamos para probar un elemento contra una llave y recorrer el elemento, convengamos que es un número constante de líneas; llamemos a esa constante *c*. Por lo tanto, podrían ser necesarias hasta *ck* líneas para insertar en un sublista de *k* elementos.

Supón que en cada llamada a insert, el valor que se está insertando es menor que cada elemento en el sublista a su izquierda. Cuando llamamos insert la primera vez, *k=1*. La segunda vez, *k = 2*. La tercera vez, *k = 3*. Y así sucesivamente hasta la última vez, cuando *k = n-1*. Por lo tanto, el tiempo total que tarda la inserción en un sublista ordenado es:

$$c \cdot 1 + c \cdot 2 + c \cdot 3 + \cdots c \cdot (n-1) = c \cdot (1 + 2 + 3 + \cdots + (n-1))$$

Esa suma es una serie aritmética, excepto que va hasta *n-1* en lugar de *n*. Al usar nuestra fórmula para una serie aritmética, obtenemos que el tiempo total que tarda la inserción en el sublista ordenado es:

$$c \cdot (n-1+1)((n-1)/2) = cn^2/2 - cn/2\,.$$

Al usar notación Θ grande, descartamos el término de orden inferior *cn/2* y los factores constantes *c* y *1/2*, para obtener el resultado de que el tiempo de ejecución de la inserción, en este caso, es de $\Theta(n^2)$.

Ordenación por mezcla (Merge Sort)

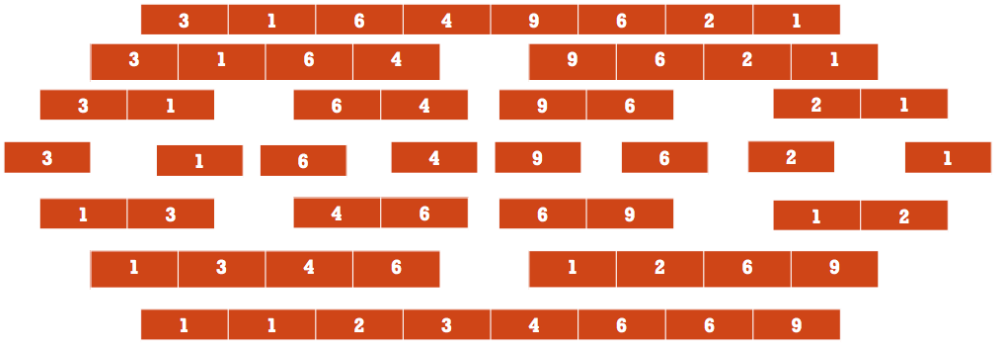
El algoritmo de ordenamiento por mezcla es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás. Es de complejidad O(*n* log *n*).

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

1. Si la longitud de la lista es 1, entonces ya está ordenada. En otro caso:
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
4. Mezclar las dos sublistas en una sola lista ordenada.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

1. Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
2. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.



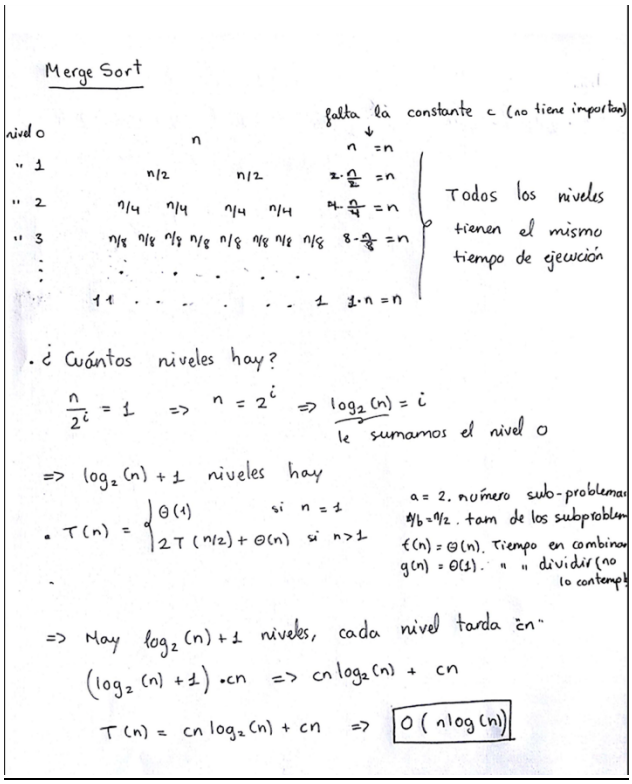
Pseudo-código

```
mergesort (vector A, int inicio, int final) {
  if (inicio < final) {
    int m = (inicio + final)/2;
    mergesort (A, inicio, m);
    mergesort (A, m+1, final);
    merge (A, inicio, m, final);
  }
}

merge (vector A, int inicio, int m, int final) {
  int i, j, K;
  vector aux[A.length];
  for i in inicio.. final
    aux[i] = A[i];
  i = inicio; j = m+1; K = inicio;
  while (i <= m && j <= final)
    if (aux[i] <= aux[j])
      A[K++] = aux[i++];
    else
      A[K++] = aux[j++];

  while (i <= m)
    A[K++] = aux[i++];
}
```

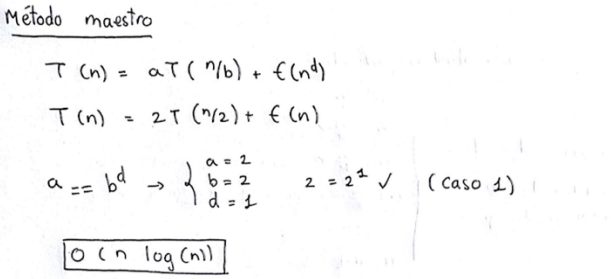
Análisis de complejidad del Merge Sort

Sabemos que la función *merge* tiene un tiempo de ejecución de O(n) al mezclar n elementos, a la vez sabemos que el proceso de dividir tiene un tiempo constante, dónde no importa el tamaño del array, ya que lo único que hace es calcular el punto medio de un array y separarlo por ese punto.

· Mediante el árbol de recurrencia



· Mediante método maestro



Comparación de Merge Sort e Insertion Sort

Como hemos podido comprobar en los análisis de complejidad de cada uno de los algoritmos el MergeSort resulta ser más rápido que el Insertion Sort debido a que el primero tiene un $O(n \cdot \log(n))$ mientras que el segundo tiene un $O(n^2)$.

Existe una optimización del método MergeSort el cual trata de que al llegar a un determinado tamaño de la sublista, se aplica el método Insertion Sort para ordenarla. Ya que se ha comprobado que para una n que sea menor de 7, el algoritmo Insertion Sort es más óptimo que el MergeSort. Sin embargo, para una n mucho mayor el algoritmo MergeSort es bastante más rápido.

```
static final int THRESHOLD = 7;
static void mergeSort(int f[],int lb, int ub){
    if (ub - lb <= THRESHOLD)
        insertionSort(f, lb, ub);
    else
    {
        int mid = (lb+ub)/2;
        mergeSort(f,lb,mid);
        mergeSort(f,mid,ub);
        merge(f,lb,mid,ub);
    }
}
```

Merge sort es a menudo la mejor opción para ordenar una lista enlazada: en esta situación es relativamente fácil implementar merge sort de manera que sólo requiera Θ(1) espacio extra, y el mal rendimiento de las listas enlazadas ante el acceso aleatorio hace que otros algoritmos (como quicksort) den un bajo rendimiento, y para otros (como heapsort) sea algo imposible.

Para Perl 5.8, merge sort es el algoritmo de ordenamiento por defecto (lo era quicksort en versiones anteriores de Perl). En Java los métodos de ordenación de Arrays usan merge sort o una modificación de quicksort dependiendo de los tipos de datos y por cuestiones de eficiencia cambian a ordenamiento por inserción cuando se están ordenando menos de siete elementos en el array.