Ordenamiento o Clasificación

El ordenamiento es algo que hacemos normalmente. Se trata de "organizar" cosas. Es simplemente colocar información de una manera especial basándonos en un criterio de ordenamiento.

En general se habla de un ordenamiento cuando existe una clave, es decir, un atributo o campo cuyo contenido identifica unívocamente un elemento. Así el criterio de ordenamiento, dependerá de la forma de clasificación que usemos, puede ser de dos formas: en forma "ascendente" cuando los elementos quedan organizados de menor a mayor, y, en forma "descendente", cuando organización de elementos es de mayor a menor.

Se han desarrollado muchas técnicas a nivel de programación para clasificar elementos, cada una con características específicas, y con ventajas y desventajas sobre las demás. Las veremos a continuación

Los algoritmos de ordenamiento se pueden clasificar de acuerdo a su comportamiento como sigue:

Estable: Si el orden relativo de los elementos con la misma clave no se altera en el proceso de ordenación

Natural: Si funciona óptimamente cuando los elementos ya están ordenados.

Antinatural: Si funciona peor cuando los elementos están ordenados

Adicionalmente es necesario evaluar el algoritmo antes de codificarlo en algún lenguaje de programación, en términos de eficiencia. Esto se hace en base a dos parámetros en función de los "n" elementos a ordenar:

a) Número de comparaciones entre claves (C)
b) Número de movimientos o transferencias de elementos (M)

La complejidad del algoritmo tiene que ver con su rendimiento, es decir, con el tiempo que tarde su ejecución. Algo así como que, si en **n** elementos se realizan **n** comparaciones la complejidad es O(n). Algunos ejemplos de complejidades comunes son:

- **O(1)** : Complejidad constante.
- **O(n^2)** : Complejidad cuadrática.
- **O(n log(n))** : Complejidad logarítmica.

Ahora podemos decir que un algoritmo de complejidad O(n) es más rápido que uno de complejidad O(n^2). Otro aspecto a considerar es la diferencia entre el peor y el mejor caso. Cada algoritmo se comporta de modo diferente de acuerdo a cómo se le entregue la información; por eso es conveniente estudiar su comportamiento en casos extremos, como cuando los datos están prácticamente ordenados o muy desordenados.

Algoritmos más usados

Existen diversos métodos de ordenación de elementos. Entre los más comunes se encuentran:

1) Ordenamiento por Selección directa
2) Ordenamiento por Inserción directa o Método de la Baraja
3) Ordenamiento po Intercambio directo o Método de la Burbuja
4) Ordenamiento por partición o "rápido" (Quicksort)

Estos algoritmos los estudiaremos a continuación asumiendo que tenemos "N" elementos a ordenar y que se encuentran en un arreglo estático llamado "A" desde la posición cero (0) hasta N-1

Ordenamiento por Selección

Descripción.

Considera TODOS los elementos de la secuencia "origen" para encontrar el que tiene la menor clave y depositarla como el elemento siguiente de la secuencia "destino"

Este algoritmo se basa en:

- Seleccionar el elemento con mínima clave
- Intercambiarlo con el primero a1 y se repite con n-1, n-2, etc.

Algoritmo Selección Directa

```
1.  INICIO
2.    Desde i = 0 hasta N-2 hacer
3.      K ← i
4.      x ← A[i]
5.      Desde j = i+1 hasta N-1 hacer
6.         Si A[j] < x entonces
7.            k ← j
8.            x ← A[j]
9.         fin_si
10.     fin_desde
11.    A[k] ← A[i]
12.    A[i]  ← x
13.   fin_desde
14. FIN
```

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.
- Realiza pocos intercambios.
- Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Código en C:

```
void seleccion (int A[])
{
    int i;
    int x, pos_men;

    for (i=0; i <N - 1; i++)
        {
        /* Buscamos el elemento menor */
        pos_men = menor(A, i);
        /* Lo colocamos en el lugar que le corresponde */
        x = A[i];
        A[i] = A [pos_men];
        A [pos_men] = x;
        }
}
```

Usa esta función para determinar el elemento menor:

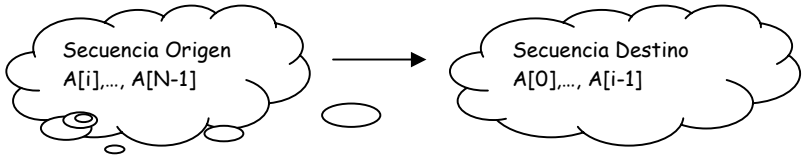
```
int menor (int A[], int desde)
{
    int i, menor, indice;

    menor = 999999;
    for (i=desde; i<N; i++)
        if (A[i] < A[menor])
        {
            menor = A[i];
            indice = i;
        }
    return indice;
}
```

Ordenamiento por Inserción o Método de la Baraja

Descripción.

Este algoritmo trabaja con una secuencia "origen" y va formando una secuencia "destino" de los elementos.



En cada paso, empezando con i = 1 e incrementando i de uno en uno, se toma al elemento i del origen y se transfiere al destino en orden.

Considera en cada paso, un ÚNICO elemento de la secuencia "origen" y TODOS los de la secuencia "destino" para encontrar el punto de inserción.

Algoritmo Inserción Directa

```
1.  INICIO
2.    Desde i = 1 hasta N-1 hacer
3.      x ← A[i]
4.      Desde j = 0 hasta i-1 hacer
5.         Si x < A[j]  entonces
6.             intercambio (A[j], x)
7.         fin_si
8.      fin_desde
9.     A[i]  ← x
10.   fin_desde
11. FIN
```

Ventajas:

- Fácil implementación.
- Requerimientos mínimos de memoria.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Código en C:

```
void insercion (int A[])
{
    int i, j, x, y;

    for (i=1; i < n + 1; i++)
        {
        x = A[i];
        /* Desplazamos los elementos mayores que A[i] */
         for (j=0; j < i; j++)
            if  ( x < A[j] )
            {
               y = A[j];
               A[j] = x;
               x = y:
            }
        /* Copiamos A[i] en su posición final */
        A[i] = x;
        }


}
```

Ordenamiento por Intercambio o Método de la Burbuja (Bubblesort)

Descripción.

Se basa en comparar e intercambiar pares de elementos adyacentes hasta que todos estén ordenados

Algoritmo Burbuja

```
1. INICIO
2.    Desde i = 1 hasta N-1 hacer
3.       Desde j = n-1 hasta i hacer (decrementando j)
4.          Si  A[j-1] > A[j] entonces
5.              intercambio (A[j-1], A[j])
6.          fin_si
7.       fin_desde
8.    fin_desde
9. FIN
```

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.

Desventajas:

- Muy lento.
- Realiza numerosas comparaciones.
- Realiza numerosos intercambios.

Este algoritmo es uno de los más pobres en rendimiento. No es recomendable usarlo.

Además tiene una peculiaridad, que lo hace, se podría decir "asimétrico": un solo elemento mal situada en el extremo "pesado" de un arreglo ordenado ya, se situará en posición correcta en una sola pasada. Por ejemplo, el 06 en la secuencia: 12-18-42-44-55-67-94-06

Sin embargo, un elemento mala colocado en el extremo "ligero" se "hunde" hacia su posición correcta a un ritmo de una posición por cada pasada. Por ejemplo, el elemento 94 en la secuencia: 94-06-12-18-42-44-55-67 requiere siete (7) pasadas.

Código en C:

```
void burbuja (int A[])
{
    int i, j;
    int x;

    for (i=1; i<N; i++)
        for (j=0; j<N - i; j++)
            if (A[j] > A[j+1])
                {
                /* Intercambiamos */
                x = A[j];
                A[j] = A[j+1];
                A[j+1] = x;
                }
}
```

Ordenamiento Rápido (Quicksort)

Descripción.

Se basa en el principio de intercambio. Es el algoritmo más eficiente para clasificación interna. Fue desarrollada por C.A.R. Hoare en 1960. El algoritmo original es recursivo, pero también se utilizan versiones iterativas.

La esencia del método consiste en clasificar un arreglo A[0],…A[N-1], tomando un elemento de división o elemento pivote "v" en la posición "k", para "reorganizar" los elementos de tal manera, que, los registros con clave menor que "v" estén en A[0],…..A[k-1] y, los elementos con clave mayor a "v" se encuentren en el intervalo A[k+1],….A[N-1].

Algoritmo Quicksort

1. INICIO
2. Establecer el elemento pivote "v" = A[k] (j arbitrario)
3. Mientras la división no se haya terminado hacer
4. Recorrer de izquierda a derecha hasta encontrar un elemento A[i] ≥ v
5. Recorrer de derecha a izquierda hasta encontrar un elemento A[j] ≤ v
6. Si los valores localizados "no se han cruzado" con "v"
7. Intercambiar los valores A[i] y A[j]
8. fin_si
9. fin_mientras
10. FIN

Explicado con detalle sería:

a) Recorres la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).
b) Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias.
c) Repites esto hasta que se crucen los índices.
d) El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

Ventajas:

- Muy rápido
- No requiere memoria adicional.

Desventajas:

- Implementación un poco más complicada.
- Recursividad (utiliza muchos recursos).
- Funciona "moderadamente" bien para pequeños valores de N
- Mucha diferencia entre el peor y el mejor caso, dependiendo del elemento de comparación o pivote seleccionado.

En nuestro algoritmo, seleccionamos el elemento medio como pivote, pero también puede seleccionarse el primero o el último elemento. En estos casos, la situación más desfavorable se produce con el arreglo ordenado ya inicialmente (antinatural). Sin embargo, si se toma el elemento medio, se comporta en forma óptima. En otras palabras, este método puede resultar excelente en algunos casos, pero en otros, muy desafortunado.

Código en C:

```
void quicksort  (int A[], iz, int de)
{
/* vect: vector, iz: índice izquierdo, de: índice  derecho */

register int i, j; /* variables índice del vector */
  int elem; /* contiene un elemento del vector */

  i = iz;
  j = de;
  elem = A[(iz+de)/2];

  do
   {
     while (A[i] < elem && j < de) /* recorrido del vector  hacia la derecha */
       i++;
     while (elem < A[j] && j > iz) /* recorrido del vector hacia la izquierda */
       j--;
     if (i <= j) /* intercambiar */
      {
        int aux; /* variable auxiliar */
        aux = A[i];
        A[i] = A[j];
        A[j] = aux;
        i++;
        j--;
      }
   } while (i <= j);
  if (iz < j)
    quicksort (A, iz, j);
  if (i < de)
    quicksort (A, i, de);
}
```

COMPARACIÓN DE LOS MÉTODOS DE ORDENAMIENTO

La siguiente es una tabla comparativa de algunos algoritmos de ordenamiento.

Nombre del Algoritmo	Complejidad (Comparaciones)		Complejidad (Movimientos)		Características
	Mínima	Máxima	Mínima	Máxima	
Selección	N-1	$((N^2 - N)/2) -1$	2 (N-1)	$(N^2 + 3N -4)/2$	Antinatural y estable
Inserción	$(N^2 - N)/2$	$(N^2 - N)/2$	3 (N-1)	$N^2/4 + 3(N-1)$	Natural y estable
Burbuja	$(N^2 - N)/2$	$(N^2 - N)/2$	0	$(N^2 - N) * 1,5$	Estable, "Asimétrico"
Quicksort	N	$N*\log_2 N$	$N/6*\log_2 N$	N^2	Depende del "pivote"

Cada algoritmo se comporta de modo diferente de acuerdo a la cantidad y la forma en que se le presenten los datos, entre otras cosas. Para seleccionar el mejor, depende del caso. Hay pistas que pueden ayudar en la selección, por ejemplo:

- ✓ Si la información va a estar casi ordenada y no quieres complicarte, un algoritmo sencillo como el ordenamiento burbuja será suficiente. Si por el contrario los datos van a estar muy desordenados, un algoritmo poderoso como Quicksort puede ser el más indicado. Y si no puedes hacer una presunción sobre el grado de orden de la información, lo mejor será elegir un algoritmo que se comporte de manera similar en cualquiera de estos dos casos extremos.
- ✓ Si la cantidad es pequeña, no es necesario utilizar un algoritmo complejo, y es preferible uno de fácil implementación. Una cantidad muy grande puede hacer prohibitivo utilizar un algoritmo que requiera de mucha memoria adicional.
- ✓ Algunos algoritmos sólo funcionan con un tipo específico de datos (enteros, enteros positivos, etc.) y otros son generales, es decir, aplicables a cualquier tipo de dato.
- ✓ Algunos algoritmos realizan múltiples intercambios (burbuja, inserción). Si los registros son de gran tamaño estos intercambios son más lentos.