Algoritmos de ordenación

1.Bubble

Merge

Fue desarrollado en 1945 por John Von Neumann.

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

1. Si la longitud de la lista es 0 ó 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.

```
/**
 * Descripción de los parámetros
 *
 * @params:
 *
 * @var inicio: Inicio del intervalo a ordenar
 * @var fin   : Fin del intervalo a ordenar
 * @var lista : La lista en la que ordenaremos el intervalo, [inicio,fin]
 *
 **/
```

```
void MergeSort(int inicio,int fin,int *lista){
    /*
    Si la sub-lista es de tamaño 1 o 0, se termina el método,
    dado que esta sub-lista ya está ordenada.
    */
    if(fin - inicio == 0 || fin - inicio == 1)
        return;

    //determinamos el punto medio del intervalo a ordenar.
    int cursor= (inicio + fin)/2;

    //Ordenamos la sub-lista de la izquierda.
    MergeSort(inicio,cursor,lista);

    //Ordenamos la sub-lista de la derecha.
    MergeSort(cursor,fin,lista);

    int puntero1 = inicio,
        puntero2 = cursor,
        puntero3 = 0;

    /*
    Creamos un arreglo donde guardaremos la mezcla
    de las sub-listas ordenadas.
    */
```

```
    int array[fin-inicio];

    //Mezclamos las sub-lista de derecha y de izquierda, en el arreglo array.
    while(puntero1<cursor || puntero2<fin){
        if(puntero1<cursor && puntero2<fin){
            if(lista[puntero1]<lista[puntero2]){
                array[puntero3++] = lista[puntero1++];
            }else{
                array[puntero3++] = lista[puntero2++];
            }
        }else if(puntero1<cursor){
            array[puntero3++] = lista[puntero1++];
        }else{
            array[puntero3++] = lista[puntero2++];
        }
    }

    /*
    Para terminar pasamos la sub-lista ordenada
    que está en el arreglo array para la lista original.
    */
    for(int i=0;i<fin-inicio;i++){
        lista[inicio+i]=array[i];
    }
}
```

A continuación, implementamos el procedimiento *Mezcla*. Aunque este procedimiento no se analizará tan minuciosamente como el del ordenamiento por inserción, debe poder verse que su complejidad es de *O(n)* donde *n=r - p + 1*, es decir el número total de elementos a *mezclar*. Esto se debe a que cada elemento de cada uno de los dos submontones, se movio al montón de resultado únicamente una vez, por lo que, si hay ***n*** elementos, el procedimiento tardará un tiempo proporcional a ***n.***

```
Mezcla(A,p,q,r)
1    n1:=q - p + 1
2    n2:=r - q
3    Crea arreglo L[1..n1 + 1]
4    Crea arreglo R[1..n2 + 1]
5    desde i:=1 hasta n1 haz
6        L[i]:=A[p + i - 1]
7    desde j:=1 hasta n2 haz
8        R[j]:=A[q + j]
9    L[n1 + 1]:=infinito
10   R[n2 + 1]:=infinito
11   i:=1
12   j:=1
13   desde k:=p hasta r haz inicio
14       Si L[i] <= R[j] entonces inicio
15           A[k]:=L[i]
16           i:=i + 1
17       Si-No entonces inicio
18           A[k]:=R[j]
19           j:=j + 1
20       fin-Si-entonces
21   fin-desde
```

En resumen, el procedimiento *Mezcla* funciona como sigue:

- Las líneas 1 y 2 calculan el largo del primer y segundo sub-arreglos.
- Las líneas 3 y 4 apartan memoria suficiente para los dos sub-arreglos.
- Los ciclos desde de las líneas 5, 6, 7 y 8 copian ambos sub-arreglos a los arreglos ***L*** y ***R***.
- En las líneas 9 y 10 se inicializa el último término de ***L*** y ***R*** a infinito, esto es muy importante, ya que aquí el infinito nos sirve para saber cuándo ya se terminó uno de los sub-arreglos. Aunque en cuestión práctica no existe tal cosa como "infinito" en las computadoras, se suele usar algún valor que previamente hayamos establecido como nuestro "infinito".
- El ciclo *desde* que inicia en la línea 13 va desde ***p*** hasta ***r***, es decir se ejecuta un número de veces igual a la suma de los elementos de ambos sub-arreglos. El funcionamiento de este ciclo es la idea básica del algoritmo. Compara el primer elemento no mezclado de cada sub-arreglo y

retira el menor, avanzando hacia el siguiente elemento del sub-arreglo de donde se retiro el elemento.

- Con el procedimiento *Mezcla* podemos implementar nuestra ordenación por mezcla. El código del algoritmo queda de la siguiente forma

```
Ordena-Mezcla(A,p,r)
1   Si p < r entonces inicio
2       q:=(p + r) div 2
3       Ordena-Mezcla(A,p,q) // ORDENA LA MITAD IZQUIERDA
4       Ordena-Mezcla(A,q+1,r) // ORDENA LA MITAD DERECHA
5                                   Mezcla(A,p,q,r)
6   fin-Si-entonces
```

2.La Ordenación de burbuja (Bubble Sort en inglés)

Es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillo de implementar.

Una manera simple de expresar el ordenamiento de burbuja en pseudocódigo es la siguiente:

```
procedimiento DeLaBurbuja (a0, a1, a2, . . . , a(n−1))
    para i ← 2 hasta n hacer
        para j ← 0 hasta n − i hacer
            si a(j) > a(j+1) entonces
                aux ← a(j)
                a(j) ← a(j+1)
                a(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```

Este algoritmo realiza el ordenamiento o reordenamiento de una lista **a** de **n** valores, en este caso de **n** términos numerados del **0** al **n-1**; consta de dos bucles anidados, uno con el índice **i**, que da un tamaño menor al recorrido de la burbuja en sentido inverso de **2** a **n**, y un segundo bucle con el índice **j**, con un recorrido desde **0** hasta **n-i**, para cada iteración del primer bucle, que indica el lugar de la burbuja.

La burbuja son dos términos de la lista seguidos, **j** y **j+1**, que se comparan: si el primero es mayor que el segundo sus valores se intercambian.

Esta comparación se repite en el centro de los dos bucles, dando lugar a la postre a una lista ordenada. Puede verse que el número de repeticiones solo depende de **n** y no del orden de los términos, esto es, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada. Esta es una característica de este algoritmo. Luego veremos una variante que evita este inconveniente.

Tenemos una lista de números que hay que ordenar:

$$a = \{55, 86, 48, 16, 82\}$$

Podemos ver que la lista que tiene cinco términos, luego:

$$n = 5$$

El índice **i** hará un recorrido de **2** hasta **n**:

que en este caso será de 2 a 5. Para cada uno de los valores de **i**, **j** tomará sucesivamente los valores de **0** hasta **n-i**:

Para cada valor de **j**, obtenido en ese orden, se compara el valor del índice **j** con el siguiente:

Si el término **j** es mayor que el término **j+1**, los valores se permutan, en caso contrario se continúa con la iteración.

3.Selection.

El ordenamiento por selección (Selection Sort en inglés) es un algoritmo de ordenamiento que requiere O x^2 operaciones para ordenar una lista de n elementos.

Descripción del algoritmo

Su funcionamiento es el siguiente:

- Buscar el mínimo elemento de la lista
- Intercambiarlo con el primero
- Buscar el siguiente mínimo en el resto de la lista
- Intercambiarlo con el segundo

Y en general:

- Buscar el mínimo elemento entre una posición *i* y el final de la lista
- Intercambiar el mínimo con el elemento de la posición *i*

De esta manera se puede escribir el siguiente pseudocódigo para ordenar una lista de *n* elementos indexados desde el 1:

```
para i=1 hasta n-1
   mínimo = i;
   para j=i+1 hasta n
      si lista[j] < lista[mínimo] entonces
         mínimo = j /* (!) */
      fin si
   fin para
   intercambiar(lista[i], lista[mínimo])
fin para
```

Este algoritmo mejora ligeramente el algoritmo de la burbuja. En el caso de tener que ordenar un vector de enteros, esta mejora no es muy sustancial, pero cuando hay que ordenar un vector de estructuras más complejas, la operación intercambiar () sería más costosa en este caso. Este algoritmo realiza muchas menos operaciones intercambiar () que el de la burbuja, por lo que lo mejora en algo. Si la línea comentada con **(!)** se sustituyera por intercambiar(lista[i], lista[j]) tendríamos una versión del algoritmo de la burbuja (naturalmente eliminando el orden intercambiar del final).

Rendimiento del algoritmo [editar]

Artículo principal: Cota ajustada asintótica

Al algoritmo de ordenamiento por selección, para ordenar un vector de **n** términos, tiene que realizar siempre el mismo número de comparaciones:

$$c(n) = \frac{n^2 - n}{2}$$

Esto es, el número de comparaciones **c(n)** no depende del orden de los términos, si no del número de términos.

$$\Theta(c(n)) = n^2$$

Por lo tanto la cota ajustada asintótica del número de comparaciones pertenece al orden de n cuadrado.

El número de intercambios **i(n)**, también es fijo, téngase en cuenta que la instrucción:

intercambiar(lista[i], lista[mínimo])

siempre se ejecuta, aun cuando **i= mínimo**, lo que da lugar:

$$i(n) = n$$

sea cual sea el vector, y el orden de sus términos, lo que implica en todos los casos un coste lineal:

$$\Theta(i(n)) = n$$

la cota ajustada asintótica del número de intercambios es lineal, del orden de n.

Asimismo, la fórmula que representa el rendimiento del algoritmo, viene dada por la función:

$$c(n) = \frac{n^2 + n}{2}$$

4.Quicksort (“divide y vencerás”)

Es la técnica de ordenamiento más rápida conocida creada por
C.Antony R Hoare en 1960

El algoritmo fundamental es el siguiente:

- ✓ Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- ✓ 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. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- ✓ 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.
- ✓ 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. Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.
- ✓ En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n).
- ✓ En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $0(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas.
- ✓ En el caso promedio, el orden es O(n·log n).

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

Pseudocódigo

```
inicio
        variables A: arreglo[1..100] entero
        variables i,j,central:entero
        variables primero, ultimo: entero
        para i = 1 hasta 100
                        leer(A[i])
                Fin para
        primero = 1
        ultimo = 100
        qsort(A[],100)
```

```
Fin


Funcion qsort(primero, ultimo:entero)
                i = primero
                j = ultimo
                central = A[(primero,ultimo) div 2]
                repetir
                                mientras A[i]central
                                                j = j - 1
                                        fin mientras
                                si i < = j
                                                aux = A[i]
                                                A[j] = A[i]
                                                A[i] = aux
                                                i = i + 1
                                                j = j - 1
                                        fin si
                        hasta que i > j
                si primero < j
                                partir(primero,j)
                        fin si
                si i < ultimo
                                partir(i, ultimo)
                        fin si
        fin funcion qsort
```