ORDENAMIENTO (SORT, SORTING, CLASIFICACIÓN)

Su finalidad es organizar ciertos datos (normalmente arreglo de elementos) en un orden creciente o decreciente mediante una regla prefijada (numérica, alfabética...). Atendiendo al tipo de elemento que se quiera ordenar puede ser:

- Ordenación interna: Los datos se encuentran en memoria (sean estos arrays, listas, etc.) y son de acceso aleatorio o directo (se puede acceder a un determinado elemento sin pasar por los anteriores).

- Ordenación externa: Los datos están en un dispositivo de almacenamiento externo (archivos) y su ordenación es más lenta que la interna.

Ordenación interna

Los métodos de ordenación interna se aplican principalmente a arreglos unidimensionales.

Los principales algoritmos de ordenación interna son:

Selección: Este método consiste en buscar el elemento más pequeño del arreglo y ponerlo en primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segundo lugar, y así sucesivamente hasta colocar el último elemento. Por ejemplo, si tenemos la lista [40, 21, 4, 9, 10, 35], los pasos a seguir son:

[4,21,40,9,10,35] <-- Se coloca el 4, el más pequeño, en primera posición : se cambia el 4 por el 40.
[4,9,40,21,10,35] <-- Se coloca el 9, en segunda posición: se cambia el 9 por el 21.
[4,9,10,21,40,35] <-- Se coloca el 10, en tercera posición: se cambia el 10 por el 40.
[4,9,10,21,40,35] <-- Se coloca el 21, en tercera posición: ya está colocado.
[4,9,10,21,35,40] <-- Se coloca el 35, en tercera posición: se cambia el 35 por el 40.

Si el arreglos tiene N elementos, el número de comprobaciones que hay que hacer es de N*(N-1)/2, luego el tiempo de ejecución está en $O(N^2)$

 # SELECCIÓN

```
def ord_seleccion(lista):
    for i in range(len(lista) - 1):
        menor = i
        for j in range(i + 1, len(lista)):
            if lista[j] < lista[menor]:
                menor = j
        if menor != i:
            intercambia(lista, menor, i)
```

Burbuja: Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, [40,21,4,9,10,35]:

Primera pasada:
[21,40,4,9,10,35] <-- Se cambia el 21 por el 40.
[21,4,40,9,10,35] <-- Se cambia el 40 por el 4.
[21,4,9,40,10,35] <-- Se cambia el 9 por el 40.
[21,4,9,10,40,35] <-- Se cambia el 40 por el 10.
[21,4,9,10,35,40] <-- Se cambia el 35 por el 40.

Segunda pasada:
[4,21,9,10,35,40] <-- Se cambia el 21 por el 4.
[4,9,21,10,35,40] <-- Se cambia el 9 por el 21.
[4,9,10,21,35,40] <-- Se cambia el 21 por el 10.

Ya están ordenados, pero para comprobarlo habría que terminar esta segunda comprobación y hacer una tercera.

Si el arreglos tiene N elementos, para estar seguro de que el arreglos está ordenado, hay que hacer N-1 pasadas, por lo que habría que hacer (N-1)*(N-i-1) comparaciones, para cada i desde 1 hasta N-1. El número de comparaciones es, por tanto, N(N-1)/2, lo que nos deja un tiempo de ejecución, al igual que en la selección, en $O(N^2)$.

 # BURBUJA

```
def ord_burbuja(lista):
    n = len(lista)
    for i in range(n-1):
        for j in range(n-1-i):
            if lista[j] > lista[j+1]:
                intercambia(lista, j, j+1)
```

Inserción directa: En este método lo que se hace es tener una sublista ordenada de elementos del arreglos e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada. Para el ejemplo [40,21,4,9,10,35], se tiene:

[40,21,4,9,10,35] <-- La primera sublista ordenada es [40].

Insertamos el 21:
[40,40,4,9,10,35] <-- aux=21;
[21,40,4,9,10,35] <-- Ahora la sublista ordenada es [21,40].

Insertamos el 4:
[21,40,40,9,10,35] <-- aux=4;
[21,21,40,9,10,35] <-- aux=4;
[4,21,40,9,10,35] <-- Ahora la sublista ordenada es [4,21,40].

Insertamos el 9:
[4,21,40,40,10,35] <-- aux=9;
[4,21,21,40,10,35] <-- aux=9;
[4,9,21,40,10,35] <-- Ahora la sublista ordenada es [4,9,21,40].

Insertamos el 10:
[4,9,21,40,40,35] <-- aux=10;
[4,9,21,21,40,35] <-- aux=10;
[4,9,10,21,40,35] <-- Ahora la sublista ordenada es [4,9,10,21,40].

Y por último insertamos el 35:
[4,9,10,21,40,40] <-- aux=35;
[4,9,10,21,35,40] <-- El array está ordenado.

En el peor de los casos, el número de comparaciones que hay que realizar es de N*(N+1)/2-1, lo que nos deja un tiempo de ejecución en $O(N^2)$. En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(N).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Vemos que cuanto más ordenada esté inicialmente más se acerca a O(N) y cuanto más desordenada, más se acerca a $O(N^2)$.

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

 # INSERCION

```
def ord_insercion(lista):
    for i in range(1, len (lista)) :
        elemento = lista[i]
        j = i - 1
        while j >= 0 and elemento < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = elemento
```

Inserción binaria: Es el mismo método que la inserción directa, excepto que la búsqueda del orden de un elemento en la sublista ordenada se realiza mediante una búsqueda binaria (ver algoritmos de búsqueda), lo que en principio supone un ahorro de tiempo. No obstante, dado que para la inserción sigue siendo necesario un desplazamiento de los elementos, el ahorro, en la mayoría de los casos, no se produce, si bien hay compiladores que realizan optimizaciones que lo hacen ligeramente más rápido.

 # Escribir una versión del algoritmos de ordenamiento llamado INSERCION BINARIA

Shell: Es una mejora del método de inserción directa, utilizado cuando el arreglo tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (llamado salto) a su izquierda. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, lo pasos para ordenar el arreglos [40,21,4,9,10,35] mediante el método de Shell serían:

Salto=3:
  Primera pasada:
  [9,21,4,40,10,35] <-- se intercambian el 40 y el 9.
  [9,10,4,40,21,35] <-- se intercambian el 21 y el 10.
Salto=1:
  Primera pasada:
  [9,4,10,40,21,35] <-- se intercambian el 10 y el 4.
  [9,4,10,21,40,35] <-- se intercambian el 40 y el 21.
  [9,4,10,21,35,40] <-- se intercambian el 35 y el 40.
  Segunda pasada:
  [4,9,10,21,35,40] <-- se intercambian el 4 y el 9.

Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

 # SHELL

```python
def ord_shell(lista):
    n = len(lista)
    gap = n//2
    while gap > 0:
        for i in range(gap,n):
            temp = lista[i]
            j = i
            while j >= gap and lista[j-gap] > temp:
                lista[j] = lista[j-gap]
                j -= gap
            lista[j] = temp
        gap //= 2
```

Ordenación rápida (quicksort): Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo la lista en listas más pequeñas, y ordenar éstas. Para hacer esta división, se toma un valor del arreglo como pivote, y se mueven todos los elementos menores que este pivote a su izquierda y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el arreglo.

Normalmente se toma como pivote el primer elemento del arreglo, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran. Por ejemplo, para dividir el arreglo [21, 40, 4, 9, 10, 35], los pasos serían:

[21, 40, 4, 9, 10, 35] <-- se toma como pivote el 21. La búsqueda de izquierda a derecha encuentra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote.
Se intercambian:
[21,10,4,9,40,35] <-- Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el pivote en su lugar (en el número encontrado por la segunda búsqueda, el 9, quedando:

[9, 10, 4, 21, 40, 35] <-- Ahora tenemos dividido el arreglo en dos arreglos más pequeños: el [9, 10, 4] y el [40, 35], y se repetiría el mismo proceso.

La implementación es claramente recursiva.

Intercalación: no es propiamente un método de ordenación, consiste en la unión de dos arreglos ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arreglos de izquierda a derecha e ir tomando el menor de los dos elementos, de forma que sólo aumenta el contador del arreglo del que sale el elemento siguiente para el arreglo-suma. Si quisiéramos sumar los arreglos [1, 2, 4] y [3, 5, 6], los pasos serían:

Inicialmente: i1 = 0, i2 = 0, is = 0.
Primer elemento: mínimo entre 1 y 3 es 1. suma=[1]. i1 = 1, i2 = 0, is = 1.
Segundo elemento: mínimo entre 2 y 3 es 2. suma=[1, 2]. i1 = 2, i2 = 0, is = 2.
Tercer elemento: mínimo entre 4 y 3 es 3. suma=[1, 2, 3]. i1 = 2, i2 = 1, is = 3.
Cuarto elemento: mínimo entre 4 y 5 es 4. suma=[1, 2, 3, 4]. i1 = 3, i2 = 1, is = 4.
Como no quedan elementos del primer arreglo, basta con poner los elementos que quedan del segundo arreglo en la suma:
suma=[1, 2, 3, 4] + [5, 6] = [1, 2, 3, 4, 5, 6]

```python
def particion(lista, izquierda, derecha):
    pivote = lista[izquierda]
    while True:
        while lista[izquierda] < pivote:
            izquierda += 1
        while lista[derecha] > pivote:
            derecha -= 1
        if izquierda >= derecha:
            return derecha
        else:
            lista[izquierda], lista[derecha] = lista[derecha], lista[izquierda]
            izquierda += 1
            derecha -= 1

def ord_quick(lista, izquierda, derecha):
    if izquierda < derecha:
        indiceParticion = particion(lista, izquierda, derecha)
        ord_quick(lista, izquierda, indiceParticion)
        ord_quick(lista, indiceParticion + 1, derecha)
```

Intercalación de listas o arreglos: No es propiamente un método de ordenación, consiste en la unión de dos arreglos ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arreglos de izquierda a derecha e ir tomando el menor de los dos elementos, de forma que sólo aumenta el contador del arreglo del que sale el elemento siguiente para el arreglo-suma. Si quisiéramos sumar las listas [1, 2, 4] y [3, 5, 6], y obtener [1, 2, 3, 4, 5, 6] que pasos se deberían seguir?.

Escribir un programa que realice esta acción.