

Algoritmos_de_ordenamiento

November 3, 2022

1 Algoritmos de ordenamiento

En informática el ordenamiento de datos cumple un rol muy importante, ya sea como un **fin en sí** o como **parte de otros procedimientos más complejos**. Se han desarrollado muchas técnicas en este ámbito, cada una con características específicas, y con ventajas y desventajas sobre las demás.

Sea en informática como en matemáticas un algoritmo de ordenamiento es un algoritmo que coloca elementos de una **lista** o un **vector** en una **secuencia dada** por una **relación de orden**. El **resultado** es una **permutación** —o 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.

2 Tipos de Algoritmos

2.1 Ordenamiento de burbuja (Bubble Sort)

Bubble sort es uno de los algoritmos de clasificación más sencillos, realiza varias pasadas a través de una lista. * Comenzando con el primer elemento, compara el elemento actual con el siguiente elemento de un arreglo. * Si el elemento actual es mayor que el siguiente elemento del arreglo, intercambia estos elementos. * Si el elemento actual es menor que el siguiente elemento, muévase al siguiente elemento.

Detalle: La idea básica del ordenamiento de la burbuja es recorrer el conjunto de elementos en forma secuencial varias veces. Cada paso compara un elemento del conjunto con su sucesor ($x[i]$ con $x[i+1]$), e intercambia los dos elementos si no están en el orden adecuado. El algoritmo utiliza una bandera que cambia cuando se realiza algún intercambio de valores, y permanece intacta cuando no se intercambia ningún valor, pudiendo así detener el ciclo y terminar el proceso de ordenamiento cuando no se realicen intercambios, lo que indica que este ya está ordenado. Este algoritmo es de fácil comprensión y programación pero es poco eficiente puesto que existen $n-1$ pasos y $n-i$ comprobaciones en cada paso, aunque es mejor que el algoritmo de ordenamiento por intercambio. En el peor de los casos cuando los elementos están en el orden inverso, el número máximo de recorridos es $n-1$ y el número de intercambios o comparaciones está dado por $(n-1) * (n-1) = n^2 - 2n + 1$. En el mejor de los casos cuando los elementos están en su orden, el número de recorridos es mínimo 1 y el ciclo de comparaciones es $n-1$. La **complejidad** del algoritmo de la burbuja es $O(n)$ en el mejor de los casos y $O(n^2)$ en el peor de los casos, siendo su complejidad total $O(n^2)$.

[Animación método de la burbuja](#)

```
[ ]: def bubble_sort(vector):
    permutation = True
    iteracion = 0
    while permutation == True:
        permutation = False
        iteracion = iteracion + 1
        for actual in range(0, len(vector) - iteracion):
            if vector[actual] > vector[actual + 1]:
                permutation = True
                # Intercambiamos los dos elementos
                vector[actual], vector[actual + 1] = vector[actual + 1], ↵
                ↵vector[actual]

lista = [10, 20, 14, 2, 6, 8, 12, 9]
print('Arreglo desordenado: {0}'.format(lista))
bubble_sort(lista)
print('Arreglo ordenado en orden ascendente:{0}'.format(lista))
```

Arreglo desordenado: [10, 20, 14, 2, 6, 8, 12, 9]
Arreglo ordenado en orden ascendente:[2, 6, 8, 9, 10, 12, 14, 20]

2.2 Ordenamiento por inserción

Al igual que la ordenación por burbuja, el algoritmo de ordenación por inserción es sencillo de implementar y comprender.

- Iterar de la posición 1 (arr[1]) a la posición n (arr[n]) sobre un arreglo.
- Compare el elemento actual (clave) con su predecesor.
- Si el elemento clave es más pequeño que su predecesor, compare sus elementos anteriores.
- Mueva los elementos más grandes una posición hacia arriba para hacer espacio para el elemento intercambiado.

Divida el arreglo en partes ordenadas y no ordenadas, luego los valores de las partes no ordenadas se seleccionan y colocan en la posición correcta en la parte ordenada.

Animación método por inserción

```
[ ]: def insertion_sort(vector):
    for i in range(1,len(vector)):
        actual = vector[i]
        j = i
        #Desplazamiento de los elementos de la matriz }
        while j>0 and vector[j-1]>actual:
            vector[j]=vector[j-1]
            j = j-1
        #insertar el elemento en su lugar
        vector[j]=actual
```

```

lista = [10, 20, 14, 2, 6, 8, 12, 9]
print('Arreglo desordenado: {}'.format(lista))
insertion_sort(lista)
print('Arreglo ordenado en orden ascendente:{}'.format(lista))

```

Arreglo desordenado: [10, 20, 14, 2, 6, 8, 12, 9]
 Arreglo ordenado en orden ascendente:[2, 6, 8, 9, 10, 12, 14, 20]

2.3 Ordenamiento por Selección

El algoritmo mantiene dos subarreglos en el arreglo dado.

1. El subarreglo que ya está ordenado.
2. Subarreglo restante que no está ordenado.

En cada iteración del ordenamiento por selección, el elemento mínimo (considerando el orden ascendente) del subarreglo no ordenado se selecciona y se mueve al subarreglo ordenado.

Detalle: * Busca el elemento más pequeño de la lista. * Lo intercambia con el elemento ubicado en la primera posición de la lista. * Busca el segundo elemento más pequeño de la lista. * Lo intercambia con el elemento que ocupa la segunda posición en la lista. * Repite este proceso hasta que hayas ordenado toda la lista.

[Animación método por selección](#)

```

[ ]: def selection_sort(vector):
    nb = len(vector)
    for actual in range(0,nb):
        mas_pequeno = actual
        for j in range(actual+1,nb) :
            if vector[j] < vector[mas_pequeno] :
                mas_pequeno = j

        temp = vector[actual]
        vector[actual] = vector[mas_pequeno]
        vector[mas_pequeno] = temp

lista = [10, 20, 14, 2, 6, 8, 12, 9]
print('Arreglo desordenado: {}'.format(lista))
selection_sort(lista)
print('Arreglo ordenado en orden ascendente:{}'.format(lista))

```

Arreglo desordenado: [10, 20, 14, 2, 6, 8, 12, 9]
 Arreglo ordenado en orden ascendente:[2, 6, 8, 9, 10, 12, 14, 20]

2.4 Merge

El algoritmo de ordenamiento por mezcla (merge sort en inglés) es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás.

La idea de los algoritmos de ordenación por mezcla es dividir la matriz por la mitad una y otra vez hasta que cada pieza tenga solo un elemento de longitud. Luego esos elementos se vuelven a juntar (mezclados) en orden de clasificación.

La idea básica del algoritmo es tener dos funciones. La primera, **merge_sort**, ordena la lista, mientras que la segunda, **merge**, se encarga de intercalar los elementos de las 2 listas ordenadas que **merge** recibe como parámetros.

Veamos un ejemplo en código:

```
[ ]: def merge_sort(lista):  
  
    """  
    Lo primero es comprobar la longitud de la lista.  
    Si es menor que 2, 1 o 0, se devuelve la lista.  
    ¿Por qué? Ya está ordenada.  
    """  
    if len(lista) < 2:  
        return lista  
  
    # De lo contrario, se divide en 2  
    else:  
        middle = len(lista) // 2  
        right = merge_sort(lista[:middle])  
        left = merge_sort(lista[middle:])  
        return merge(right, left)  
  
  
def merge(lista1, lista2):  
    """  
    merge se encargara de intercalar los elementos de las dos  
    divisiones.  
    """  
    i, j = 0, 0 # Variables de incremento  
    result = [] # Lista de resultado  
  
    # Intercalar ordenadamente  
    while(i < len(lista1) and j < len(lista2)):  
        if (lista1[i] < lista2[j]):  
            result.append(lista1[i])  
            i += 1  
        else:  
            result.append(lista2[j])  
            j += 1
```

```

# Agregamos los resultados a la lista
result += lista1[i:]
result += lista2[j:]

# Retornamos el resultados
return result

# Prueba del algoritmo

lista = [31,3,88,1,4,2,42]

merge_sort_result = merge_sort(lista)
print(merge_sort_result)

```

[1, 2, 3, 4, 31, 42, 88]

2.5 Quick short

El ordenamiento rápido (tambien llamado ordenamiento de Hoare o quicksort en inglés) se basa también en la técnica de divide y vencerás. Esta es la técnica quizás la más eficiente que en la mayoría de los casos da mejores resultados.

El algoritmo fundamental es el siguiente:

1. Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
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. 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.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

Animación método quick short

```

[ ]: def quick_sort(vector):
    if not vector:
        return []
    else:
        pivote = vector[-1]
        menor = [x for x in vector if x < pivote]
        mas_grande = [x for x in vector[:-1] if x >= pivote]
        return quick_sort(menor) + [pivote] + quick_sort(mas_grande)

lista = [10, 20, 14, 2, 6, 8, 12, 9]
print(lista)

```

```
print('Arreglo ordenado en orden ascendente:{0}'.format(quick_sort(lista)))
```

[10, 20, 14, 2, 6, 8, 12, 9]

Arreglo ordenado en orden ascendente:[2, 6, 8, 9, 10, 12, 14, 20]

Referencias

http://lwh.free.fr/pages/algo/tri/tri_es.htm

<https://eiposgrados.com/blog-python/tipos-de-algoritmos-de-ordenacion-en-python/>

<https://github.com/ForeignGods/Sorting-Algorithms-Blender>