Algoritmos de ordenación

Reporte de los algoritmos vistos en clases

Paulina Aguirre García 1837503

Un algoritmo de ordenación nos permite ordenar los elementos de un arreglo o matriz, de la manera deseada. Para esto existen diferentes algoritmos, algunos con mayor o menor complejidad y eficacia.

Selection:

Este algoritmo consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenarlo todo.

Rendimiento del algoritmo: Cada búsqueda requiere comparar todos los elementos no clasificados, de manera que el número de comparaciones C(n) no depende del orden de los términos, si no del número de términos; por lo que este algoritmo presenta un comportamiento constante independiente del orden de los datos. C(n)= n(n-1)/2. Luego la complejidad es del orden Θ(n2).

Implementación en Python:

```
1. def selection_sort(arr):
2.     nb = len(arr)
3.     for actual in range(0,nb):
4.         mas_pequeno = actual
5.         for j in range(actual+1,nb) :
6.             if arr[j] < arr[mas_pequeno] :
7.                 mas_pequeno = j
8.         if min is not actual :
9.             temp = arr[actual]
10.                 arr[actual] = arr[mas_pequeno]
11.                 arr[mas_pequeno] = temp
```

Bubble:

La ordenación de burbuja 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.

Rendimiento del algoritmo:

- Rendimiento en el caso óptimo: En el caso óptimo, con los datos ya ordenados, el algoritmo sólo efectuará n comparaciones. Por lo tanto la complejidad en el caso óptimo es en $\Theta(n)$.
- Rendimiento en el caso desfavorable: n el caso desfavorable, con los datos ordenados a la inversa, la complejidad es en $\Theta(n^2)$.
- Rendimiento en el caso medio: En el caso medio, la complejidad de este algoritmo es también en $\Theta(n^2)$

Implementación en Python:

```
1. def bubble_sort(arr):
2.     permutation = True
3.     iteración = 0
4.     while permutation == True:
5.         permutation = False
6.         iteración = iteración + 1
7.         for actual in range(0, len(arr) - iteración):
8.             if arr[actual] > arr[actual + 1]:
9.                 permutation = True
10.                     # Intercambiamos los dos elementos
11.                     arr[actual], arr[actual + 1] = \
12.                     arr[actual + 1],arr[actual]
13.         return arr
```

Quicksort:

El ordenamiento rápido es un algoritmo creado por el científico británico en computación Tony Hoare y basado en la técnica de divide y vencerás. Este algoritmo es quizá el más eficiente.

El algoritmo consiste en:

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.

Rendimiento del algoritmo: 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 $O(n^2)$.

Implementación en Python:

```
1. def quick_sort(vector):
2.     if not vector:
3.         return []
4.     else:
5.         pivote = vector[-1]
6.         menor = [x for x in vector      if x <  pivote]
7.         mas_grande = [x for x in vector[:-1] if x >=
   pivote]
8.         return quick_sort(menor) + [pivote] +
   quick_sort(mas_grande)
```

Insertion:

La idea de este algoritmo de ordenación consiste en ir insertando un elemento de la lista o un arreglo en la parte ordenada de la misma, asumiendo que el primer elemento es la parte ordenada, el algoritmo ira comparando un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, insertando el elemento en la posición correcta dentro de la parte ordenada, y así sucesivamente hasta obtener la lista ordenada.

Rendimiento del algoritmo:

- Rendimiento en el caso óptimo: En el caso óptimo, con los datos ya ordenados, el algoritmo sólo efectuara *n* comparaciones. Por lo tanto la complejidad en el caso óptimo es en Θ(*n*).
- Rendimiento en el caso desfavorable: En el caso desfavorable, con los datos ordenados a la inversa, se necesita realizar *(n-1) + (n-2) + (n-3) .. + 1* comparaciones e intercambios, o (n2-n) / 2. Por lo tanto la complejidad es en Θ(n^2).
- Rendimiento en el caso medio: En el caso medio, la complejidad de este algoritmo es también en Θ(n^2).

Implementación en Python:

```
1. def Insertion_sort(arr):
2.     for i in range(1,len(arr)):
3.         actual = arr[i]
4.         j = i
5.         #Desplazamiento de los elementos de la matriz
6.         while j>0 and arr[j-1]>actual:
7.             arr[j]=arr[j-1]
8.             j = j-1
9.         #insertar el elemento en su lugar
10.             arr[j]=actual
```