Reporte de Algoritmos de Ordenamiento.

María Luisa Borrego Riojas

Universidad Autónoma de Nuevo León

Facultad de Ciencias Físico Matemáticas

Matemáticas Computacionales

Resumen.

En ciencias computacionales, los algoritmos de ordenamiento son utilizados para poner los elementos de una lista en un determinado orden. Como es de imaginar, existen muchas maneras diferentes de realizar este trabajo, lo que da lugar a un gran número de algoritmos diferentes. No obstante, en este reporte abarcaremos sólo cuatro de ellos: Bubblesort, Insertion Sort, Selection Sort y Quicksort. De cada uno de ellos se hablará de cómo funcionan, su complejidad y el pseudocódigo que utilizan.

Se utilizará el arreglo A = [54,26,93,17,77,31,44,55,20]

1. Bubble Sort.

Bubble sort realiza varios “chequeos” de la lista que se desea ordenar. En pocas palabras, compara el posicionamiento de los elementos del arreglo e intercambia aquellos que estén fuera de lugar. Cada vez que checa la lista, coloca el valor más grande que sigue en el lugar indicado. El nombre viene de la idea de que cada elemento sube “burbujeando” hasta la posición a la que pertenece.

Para analizar este algoritmo, debemos notar que sin importar cómo esté organizada originalmente la lista, siempre va a existir n-1 chequeos para ordenar una lista tamaño n.

Esto significa que tiene una complejidad de $\Omega(n^2)$. A continuación, el pseudocódigo para expresar el Bubble sort en Python:

```
def bubbleSort (A):
      for x in range (len(A) -1, 0, -1):
            for i in range (x):
                  if A[i] > A[i+1]
                        temp = A[i]
                        A[i] = A[i+1]
                        A[i+1] = temp
A = [54,26,93,17,77,31,44,55,20]
bubbleSort(A)
print(A)
```

A pesar de que este método es uno de los algoritmos más sencillos de implementar, también es conocido como el algoritmo más ineficiente que existe. No obstante, gracias a su simplicidad es fácil de utilizar como una introducción al concepto de algoritmo de ordenamiento para estudiantes que apenas se familiarizan con el concepto.

2. Insertion Sort.

Aunque el algoritmo insertion sort continúe siendo de complejidad $\Omega(n^2)$, funciona de una manera distinta y más eficiente que bubble sort. En este algoritmo, se cuenta con una segunda lista ordenada en las primeras posiciones de la lista principal. Cada elemento nuevo se "inserta" a esta segunda lista, por lo que el tamaño de la segunda lista siempre será un elemento más cada vez. Por ejemplo, en nuestro arreglo A = [54,26,93,17,77,31,44,55,20], la segunda lista (B) empezaría siendo de un solo elemento, en este caso B = [54], acto seguido, se inserta el elemento 2 de A, por lo tanto, B = [26,54], y así sucesivamente hasta que B se convierta en el arreglo A ya ordenado.

A continuación, el pseudocódigo para Python:

```
def insertionSort (A):
      for index in range (1,len(alist)):
            valorActual = A[index]
            posicion = index

            while posicion > 0 and A[posicion-1]>valorActual:
                  A [posicion] = A[posicion-1]
                  posicion = posicion-1

            A[posicion] = valorActual
A = [54,26,93,17,77,31,44,55,20]
insertionSort(A)
print(A)
```

Es cierto que una de las grandes desventajas del insertion sort es que pierde eficiencia cuando se trata de grandes arreglos, pero eso no quiere decir que no cuente con sus ventajas. Es un algoritmo de fácil implementación, además, es mucho más eficiente para arreglos pequeños si lo comparamos con otros algoritmos que compartan su misma complejidad, como lo son bubble sort o selection sort. También es un algoritmo adaptable, estable, sólo requiere un valor constante de Ω(1) de espacio de memoria adicional, y puede ordenar una lista conforme la va recibiendo.

3. Selection Sort.

Este tipo de algoritmo es una mejora del bubble sort al realizar solo un intercambio por cada “chequeo” de la lista. Para llevar a cabo esto, selection sort busca el mayor valor conforma pasa la lista, y después de completar este proceso, localiza el elemento en la posición adecuada. Así, el proceso continúa y requiere de n-1 pasadas para acomodar n elementos, ya que el último elemento debería estar en su lugar después de (n-1) pasadas a

la lista. Esto es, por ejemplo, en A = [54,26,93,17,77,31,44,55,20], busca el valor más grande, el cual es 93, entonces lo pasa al final: A = [54,26,20,17,77,31,44,55,93]. El valor siguiente es 77, entonces lo coloca a la izquierda del 93: A = [54,26,20,17,55,31,44,77,93]. Este proceso se repite hasta terminar de ordenar.

Ahora, el pseudocódigo:

```
def selectionSort (A):
      for x in range (len(A) -1, 0, -1):
            posicionmax = 0
            for location in range (1, fillslot+1):
                  if A[location] > A[posicionmax]:
                        posicionmax = location
            temp = A[fillslot]
            A[fillslot] = A[posicionmax]
            A[posicionmax] = temp

A = [54,26,93,17,77,31,44,55,20]
selectionSort(A)
print(A)
```

Finalmente, selection sort es superado por otros tipos de algoritmos con diferente complejidad cuando se trata de arreglos muy grandes, pero al hablar de arreglos pequeños, selection sort se caracteriza por su rapidez y eficiencia.

Se puede notar como este algoritmo realiza el mismo número de comparaciones que bubble sort, por lo que ambos tienen la misma complejidad. Sin embargo, debido a la reducción de intercambios, selection sort lo realiza más rápido, particularmente en este ejemplo, bubble sort realiza 20 intercambios, y selection sort sólo realiza 8.

4. Quicksort.

Este algoritmo para funcionar selecciona primeramente un valor pivote. Este valor puede variar, pero para explicarlo a continuación utilizaremos el primer elemento de la lista como pivote. Este valor pivote es usado para poder dividir la lista en dos, creando dos arreglos vacíos adicionales, donde uno de ellos contendrá todos los elementos que sean menores al pivote (le llamaremos m), y el otro los elementos mayores (M). Esto significa que una vez que se separa la lista, el pivote queda en el lugar al que pertenece en relación con la lista. Una vez que termina este proceso, se selecciona otro pivote para la lista m y M, respectivamente, repitiéndose el mismo procedimiento que con el arreglo original.

Ahora, el pseudocódigo:

```
      def quicksort (A)
            if len(A) <= 1
                  return A
            else:
                  return quicksort ([x for x in A[1:]if x<A[0]]) +
[A[0]]+quicksort([x for x in A[1:]if x >= A[0]])

      A = [54,26,93,17,77,31,44,55,20]
      quicksort(A)
      print(A)
```

Analizando esta función, notamos que para una lista de tamaño n, si la división ocurre siempre a la mitad de la lista, tenemos logn divisiones. Para encontrar el punto donde ocurre la división, cada elemento n necesita se comparar con el valor pivote, terminando con una complejidad de nlogn. Las ventajas de Quicksort son, primordialmente, que es muy rápido y no requiere memoria adicional.