01/09/17

Reporte: Algoritmos de ordenamiento

Matemáticas Computacionales

Ana Cecilia García Esquivel

Resumen.

En este reporte se mencionan cuatro tipos de algoritmos de ordenamiento en python: bubble, insertion, selection y quick sort, los cuales se trabajaron en clase. Acerca de ellos, se menciona en qué consisten, cómo trabajan, su complejidad, entre otras cosas.

1. Bubble

Este algoritmo sirve para ordenar una lista de elementos, generalmente de menor a mayor, pero se puede modificar para ordenar algo de diferente manera si así se quiere.

Lo que hace este algoritmo es que compara a dos elementos que estén seguidos (sin importar si son continuos) dentro de la lista, luego revisa cuál es mayor y si el primer elemento de los dos elegidos es mayor que el segundo, entonces los intercambia, y luego sigue haciendo lo mismo con los demás, por ejemplo: |6| 4 |1|2|5| → 6>4→ |4|6|→ |4|6|1|2|5|→ 6>1→ |1|6| → |4|1|6|2|5|, y así sucesivamente, hasta que el |6| esté ubicado en la última posición. Luego repite el proceso desde el principio. Ahora bien, si de los dos números elegidos el primero es menor que el segundo, entonces se quedan igual y se compara el segundo número con el tercero, y se sigue el proceso anterior.

Este algoritmo es muy poco recomendable para listas muy largas ya que se hacen 'n' operaciones para 'n' elementos, o sea tiene una complejidad de n^2.

A continuación, un pseudocódigo del algoritmo:

```
def orden(E):
    n=len(E)
    score=0
    for i in E:
        score=score+i
    score=float(score)/n
    return score

def ordenamiento(L):
    n=len(L)
    for i in range(1, n):
        for j in range(n-i):
            if orden(L[j])>orden(L[j+1]):
                L[j], L[j+1]=L[j+1], L[j]
    return L

# y ordenamos
Lista2=ordenamiento(Lista2)
```

2. **Insertion**

Este algoritmo lo que hace es seleccionar un elemento de la lista, empezando por el segundo elemento, y luego lo compara uno por uno con los elementos que estén antes del mismo, luego lo "inserta" en el lugar correspondiente y los de la derecha se recorren. Esto se repite por n- elementos de la lista. Prácticamente lo que hace es comparar los elementos e irlos dejando ordenados a la izquierda, tal que al llegar al último elemento ya estén todos los demás elementos ordenados, y ya sólo los ubica donde deben estar.

Ordenar por inserción una lista de tamaño N puede insumir (en el peor caso) tiempo del orden de N^2. En cuanto al espacio utilizado, nuevamente sólo se tiene en memoria la lista que se desea ordenar y algunas variables de tamaño 1.

A continuación un pseudocódigo:

```
def ord_insercion(lista):
    """ Ordena una lista de elementos según el método de inserción.
        Pre: los elementos de la lista deben ser comparables.
        Post: la lista está ordenada. """

    # i va desde la primera hasta la penúltima posición de la lista
    for i in xrange(len(lista)-1):
        # Si el elemento de la posición i+1 está desordenado respecto
        # al de la posición i, reubicarlo dentro del segmento (0:i]
        if lista[i+1]< lista[i]:
            reubicar(lista, i+1)

        print "DEBUG: ", lista

def reubicar(lista, p):
    """ Reubica al elemento que está en la posición p de la lista
        dentro del segmento (0:p-1].
        Pre: p tiene que ser una posicion válida de lista. """

    # v es el valor a reubicar
    v = lista[p]

    # Recorre el segmento (0:p-1] de derecha a izquierda hasta
    # encontrar la posición j tal que lista(j-1] <= v < lista(j].
    j = p
    while j > 0 and v < lista[j-1]:
        # Desplaza los elementos hacia la derecha, dejando lugar
        # para insertar el elemento v donde corresponda.
        lista[j] = lista[j-1]
        # Se mueve un lugar a la izquierda
        j -= 1

    # Ubica el valor v en su nueva posición
    lista[j] = v
```

3. **Selection**

El algoritmo de ordenación por selección se basa en la selección sucesiva de los valores mínimos o máximos. Supongamos que tenemos una lista que queremos ordenar en orden ascendente (de menor a mayor). El elemento más pequeño será el primero de la lista, y el elemento más grande será el último de la lista. Lo primero que hacemos es encontrar el valor mínimo de la lista. Después de encontrar el valor mínimo, movemos dicho valor al primer elemento en la lista. Ahora que estamos seguros de que el primer elemento está en la posición correcta de la lista, repetimos el paso anterior (encontramos el valor mínimo) a partir del segundo elemento de la lista. En este punto, estamos seguros de que el primer elemento y el segundo elemento están en sus posiciones correctas. Así que ahora estamos seguros de que los tres primeros elementos se encuentran en las posiciones correctas. Para los demás elemento se repetiría el proceso anterior.

Este es un ejemplo del algoritmo selection:

```
def selectionSort(aList):
    for i in range(len(aList)):
        least = i
        for k in range(i+1, len(aList)):
            if aList[k] < aList[least]:
                least = k

        swap(aList, least, i)

def swap(A, x, y):
    temp = A[x]
    A[x] = A[y]
    A[y] = temp
```

4. **Quicksort**

Este es el más famoso de los algoritmos recursivos de ordenamiento. Su fama radica en que en la práctica, con casos reales, es uno de los algoritmos más eficientes para ordenar. Este método se basa en la siguiente idea: Si la lista es pequeña (vacía o de tamaño 1) ya está ordenada y no hay nada que hacer. De lo contrario hacer lo siguiente: Tomar un elemento de la lista (por ejemplo el primero) al que llamaremos pivote y armar a partir de esa lista tres sublistas: la de todos los elementos de la lista menores al pivote, la formada sólo por el pivote, y la de los elementos mayores o iguales al pivote, pero sin contarlo al pivote. Ordenar cada una de esas tres sublistas (usando este mismo método). Concatenar las tres sublistas ya ordenadas.

El costo en cuanto a tiempo del quick sort se puede medir con la sig. ecuación: $T(N) = N * \log_2 N$. Lo cual es mucho menor que los anteriores tipos de algoritmo.

El siguiente puede ser un pseudo código del algoritmo:

```
def quick_sort(lista):
  """ Ordena la lista de forma recursiva.
      Pre: los elementos de la lista deben ser comparables.
```

```
        Devuelve: una nueva lista con los elementos ordenados. """

    # Caso base
    if len(lista) < 2:
      return lista
    # Caso recursivo
    menores, medio, mayores = _partition(lista)
    return quick_sort(menores) + medio + quick_sort(mayores)

def _partition(lista):
    """ Pre: lista no vacía.
        Devuelve: tres listas: menores, medio y mayores. """

    pivote = lista[0]
    menores = []
    mayores = []
    for x in xrange(1, len(lista)):
      if lista[x] < pivote:
        menores.append(lista[x])
      else:
        mayores.append(lista[x])
    return menores, [pivote], mayores
```

Conclusiones.

De todos los algoritmos de ordenamiento indicados el que facilita mejor una tarea real es el quicksort, por las razones antes mencionadas, y aunque yo no los entiendo aún muy bien, fue el que pude entender un poco mejor, al igual que el insertion. Espero pronto aprender bien y poderlos usar de una manera correcta.