Algoritmos de Ordenamiento.

Selection (Selección).

¿Qué hace?

Su funcionamiento es el siguiente:

-Buscar el mínimo elemento de la lista.

-Intercambiarlo con el primero.

-Buscar el siguiente mínimo en el resto de la lista.

-Intercambiarlo con el segundo.

Y de manera general lo que hace es buscar el mínimo elemento entre una posición i y el final de la lista. Intercambiar el mínimo con el elemento de la posición i.

¿Cómo lo hace?

Ejemplo:

```
def selectionsort(arr):
        n=len(arr)
        for i in range(0,n-1):
            posicion=i
            valor=arr[j]:
            for j in range(i,n):
                if (arr[j]<valor):
                    valor=arr[j]
                    posicion=j
            arr[posicion]=arr[i]
            arr[i]=valor
        return arr
```

¿Qué tan eficiente es?

Es lento y poco eficiente cuando se usa en listas grandes o medianas. Realiza numerosas comparaciones. Aunque es fácil su implementación y no requiere memoria adicional. Realiza pocos intercambios, además tiene un rendimiento constante, pues existe poca diferencia entre el peor y el mejor caso.

Bubble (Burbuja):

¿Qué hace?

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. A este algoritmo se le llama asi ya que la forma con la que suben por la lista los elementos durante los intercambios es como si fueran burbujas pequeñas.

¿Cómo lo hace?

Ejemplo:

```
def Burbuja (A):
        for i in range (0, len(A)-1):
        for j in range(0,(len(A)-1):
                if(A[j+1]<A[j]):
                        aux=A[j]:
                        A[j]=A[j+1]:
                        A[j+1]=aux
```

¿Qué tan eficiente es?

Este algoritmo es muy deficiente ya que al ir comparando las casillas para buscar el siguiente más grande, éste vuelve a comparar las ya ordenadas. A pesar de ser el algoritmo de ordenamiento más deficiente que hay, éste es el más usado en todos los lenguajes de programación.

Quiksort (Ordenamiento rápido):

¿Qué hace?

Primero elige un elemento de la lista de elementos a ordenar, al que se le llama “pivote”.

Acomoda 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, los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de lo que manera que quieras hacerlo. En ese momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.

La lista queda separada en dos “sublistas”, uno con los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

Se repite este proceso varias veces para cada sublista mientras éstas tengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

¿Cómo lo hace?

Ejemplo:

```
def quicksort(L):
    //Se elige el pivote
    if len(L) > 2: v = L[0] + L[len(L)-1] + L[len(L)-1/2] / 3
    elif len(L) == 2: v = L[0] + L[1] / 2
    else: v = L[0]

    return quicksort(filter((lambda y: y<v ), L)) + [v] + quicksort(filter((lambda y: y>=v), L))
```

¿Qué tan eficiente es?

La velocidad de Quiksort depende en gran medida de cómo se implementa este algoritmo, una mala implementación puede suponer que el algoritmo se ejecute a una velocidad mediocre o incluso pésima, también la elección del pivote determina las particiones de la lista de datos, es decir, esta es la parte más crítica de la implementación del algoritmo QuickSort.

Insertion (Inserción):

¿Qué hace?

Este algoritmo va comparando los datos que recibe con el primero recibido, a partir de ello los acomoda de menor a mayor, analizándolos de izquierda a derecha.

¿Cómo lo hace?

Ejemplo:

```
def insert_sort(lista):
return insert_sort_aux(lista,1,len(lista))
def insert_sort_aux(lista,i,n):
if i == n:
return lista
aux = lista[i]
j = incluye_orden(lista,i,aux)
lista[j] = aux
```

```
return insert_sort_aux(lista,i+1,n)

def incluye_orden(lista,j,aux):

if j <= 0 or lista[j-1] <= aux:

return j

lista[j] = lista[j-1]

return incluye_orden(lista,j-1,aux)
```

¿Qué tan eficiente es?

Este algoritmo por inserción es muy estable, ocupa poca memoria en comparación con otros, pero es más lento al procesar demasiada información.

Matemáticas Computacionales. Grupo:001

Ruiz Fraser Francisco Guillermo. 1837506