Rafael Enrique Almaraz Martínez 18/09/2017

Reporte: Algoritmos de ordenamiento

Bubble: Este algoritmo de ordenamiento realiza un cierto número de vueltas al arreglo de números comparándolos por parejas acorde a como sea requerido, de menor a mayor o de mayor a menor, sin embargo es un tanto ineficiente porque terminaría realizando un numero exagerado de operaciones.

Prodecimiento DeLaBurbuja (a0,a1,a2,…a(n-1))

Para i<2 hasta n hacer

Para j<0 hasta n-i hacer

Si aj >aj+1 entonces

aux<aj

aj<aj+1

aj+1<aux

fin si

fin para

fin para

fin procedimiento

|  |
| --- |
| contador = 0 |
|  |  |
|  | def burbuja(arr): |
|  | aux=arr[:] |
|  | global contador |
|  | for i in range(len(arr)): |
|  | for j in range(0,len(arr)-i-1): |
|  | if(aux[j]>aux[j+1]): |
|  | aux[j],aux[j+1]=aux[j+1],aux[j] |
|  | contador+=1 |
|  | return aux |
|  |  |
|  | import random |
|  | r = random.sample(range(0,1000),200) |
|  | print(r) |
|  | rsorted=burbuja(r) |
|  | print(contador) |
|  | print(rsorted) |

Insertion: A diferencia de bubble, insertion en una sola vuelta acomoda los términos del arreglo ya que las comparaciones las ejecuta en conjuntos cada vez mayores empezando por parejas hasta llegar al tamaño total del arreglo.

1. Llama insert para insertar el elemento que comienza en el índice 1 en el índice 0 del subarreglo ordenado.
2. Llama insert para insertar el elemento que comienza en el índice 2 en los índices del 0 al 1 del subarreglo ordenado .
3. Llama insert para insertar el elemento que comienza en el índice 3 en los índices del 0 al 2 del subarreglo ordenado .
4. Por último, llama insert para insertar el elemento que comienza en el índice n-1*n*−1n, minus, 1 en los índices del 0 al n-2*n*−2n, minus, 2 del subarreglo ordenado.

|  |
| --- |
| contador=0 |
|  | def insercion(arr): |
|  | global contador |
|  | for numero in range(1,len(arr)): |
|  | value=arr[numero] |
|  | a=numero-1 |
|  | while a>=0: |
|  | contador+=1 |
|  | if value<arr[a]: |
|  | arr[a+1]=arr[a] |
|  | arr[a]=value |
|  | a-=1 |
|  | else: |
|  | break |
|  | return arr |
|  |  |
|  |  |
|  | import random |
|  | r = random.sample(range(0,1000),200) |
|  | print(r) |
|  | rsorted=insercion(r) |
|  | print(contador) |
|  | print(rsorted) |

Selection: El algoritmo de selección identifica el valor, ya sea menor o mayor, del arreglo y lo coloca en la primera posición que le corresponde, asi continuando con los demás elementos hasta que el listado este en el orden deseado.

**para** i=1 **hasta** n-1

mínimo = i;

**para** j=i+1 **hasta** n

**si** lista[j] < lista[mínimo] **entonces**

mínimo = j **/\* (!) \*/**

**fin si**

**fin para**

intercambiar(lista[i], lista[mínimo])

**fin para**

|  |
| --- |
| contador=0 |
|  |  |
|  | def selection(arr): |
|  | aux=arr[:] |
|  | global contador |
|  | for i in range(0,len(arr)-1): |
|  | value=i |
|  | for j in range(i+1,len(arr)): |
|  | contador=contador+1 |
|  | if arr[j]<arr[value]: |
|  | value=j |
|  | if value!=i: |
|  | aux=arr[i] |
|  | arr[i]=arr[value] |
|  | arr[value]=aux |
|  | return contador |
|  |  |
|  | import random |
|  | r = random.sample(range(0,1000),200) |
|  | print(r) |
|  | rsorted=selection(r) |
|  | print(contador) |
|  | print(rsorted) |

Quicksort: De manera más compleja pero más eficiente, quicksort escoge un valor cualquiera del arreglo para convertirlo en su pivote, una vez obtenido el pivote comparar el resto de los elementos del conjunto con este para situarlos a ambos lados, ya sean menores o mayores.

inicio

variables A: arreglo[1..100] entero

variables i,j,central:entero

variables primero, ultimo: entero

para i = 1 hasta 100

leer(A[i])

Fin para

primero = 1

ultimo = 100

qsort(A[],100)

Fin

Funcion qsort(primero, ultimo:entero)

i = primero

j = ultimo

central = A[(primero,ultimo) div 2]

repetir

mientras A[i]central

j = j - 1

fin mientras

si i < = j

aux = A[i]

A[j] = A[i]

A[i] = aux

i = i + 1

j = j - 1

fin si

hasta que i > j

si primero < j

partir(primero,j)

fin si

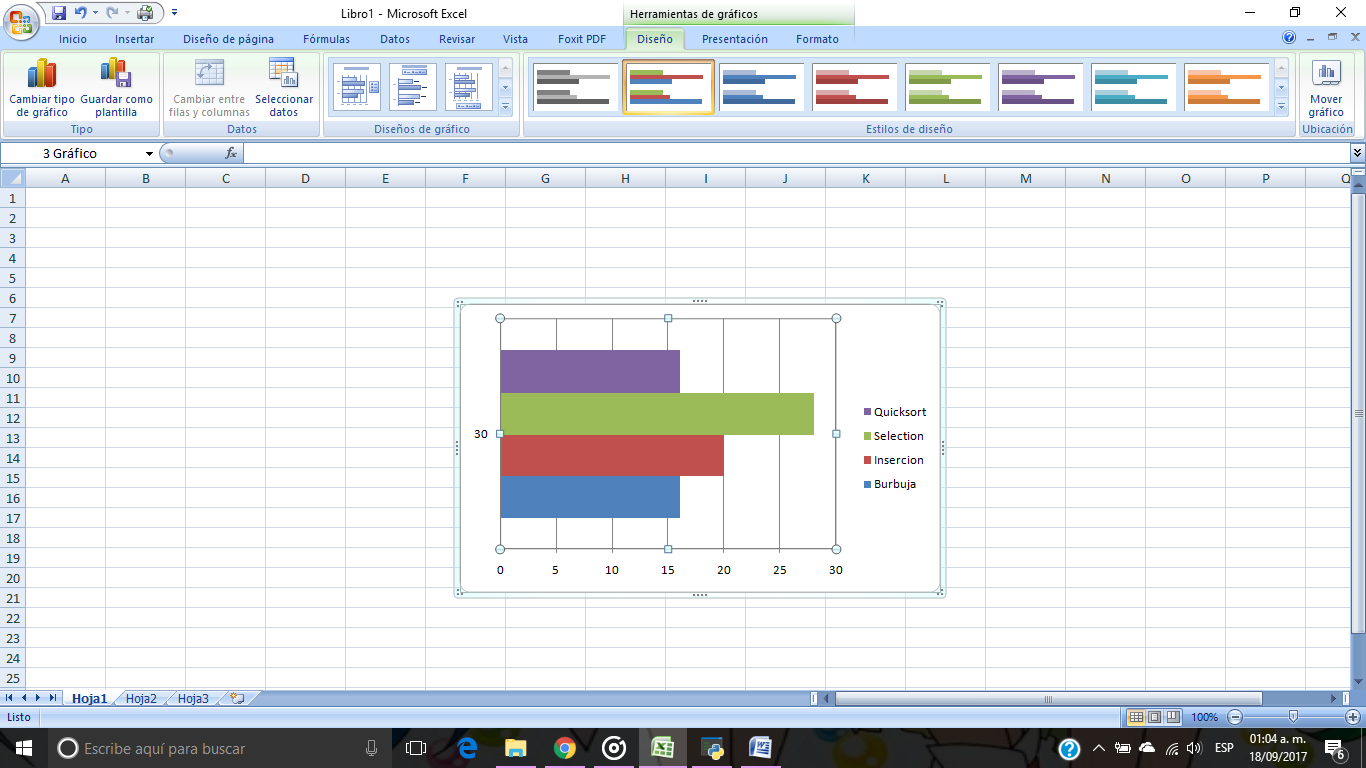
si i < ultimo

partir(i, ultimo)

fin si

fin funcion qsort

|  |
| --- |
| contador = 0 |
|  |  |
|  | def quicksort(array): |
|  | global contador |
|  | if array==[] : |
|  | return [] |
|  | m=array[0] |
|  | left=[] |
|  | right=[] |
|  | for k in array[1:]: |
|  | if k<m: |
|  | left.append(k) |
|  | else: |
|  | right.append(k) |
|  | contador+=1 |
|  | return quicksort(left)+[m]+quicksort(right) |
|  |  |
|  | import random |
|  | r = random.sample(range(0,1000),200) |
|  | print(r) |
|  | rsorted=quicksort(r) |
|  | print(contador) |
|  | print(rsorted) |



Conclusión: En base a la gráfica presentada aquí, basada en una pequeña prueba de un simple arreglo [6,5,3,1,8,7,2,4] podríamos decir que el algoritmo que puede hacer menos operaciones y por ende el mas eficaz de los 4, resulta ser el quicksort, quien a pesar de estar empatado con burbuja, su algoritmo general nlogn concluye haciendo menos operaciones que n^2 en la mayoría de los casos.