

Ordenamiento por Burbuja.

1. Descripción.

^
—

Este es el algoritmo más sencillo probablemente. Ideal para empezar. Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian.

2. Análisis del algoritmo.

^
—

Éste es el análisis para la versión no optimizada del algoritmo:

- Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto, es *estable*.
- Requerimientos de Memoria: Este algoritmo sólo requiere de una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: El ciclo interno se ejecuta **n** veces para una lista de **n** elementos. El ciclo externo también se ejecuta **n** veces. Es decir, la complejidad es $n * n = O(n^2)$. El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo $O(n^2)$.

Ventajas:

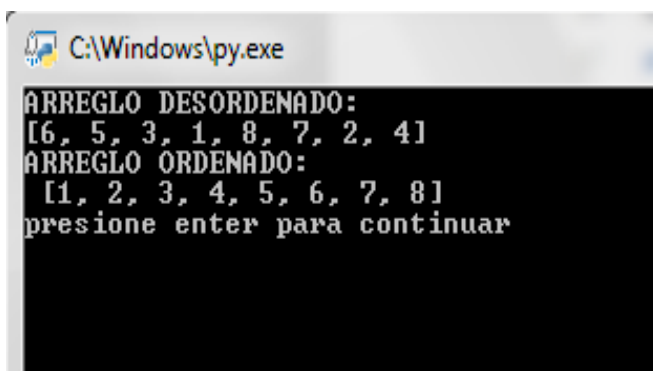
- Fácil implementación.
- No requiere memoria adicional.

Desventajas:

- Muy lento.
- Realiza numerosas comparaciones.
- Realiza numerosos intercambios.

Este algoritmo es uno de los más pobres en rendimiento. No es recomendable usarlo. Tan sólo está aquí para que lo conozcas, y porque su sencillez lo hace bueno para empezar. Ya veremos otros mucho mejores.

```
1  #Python 3.6.2
2  #Leslie Jaressy Morales Ortiz.
3  cnt=0
4  def burbuja(A):
5      global cnt
6      for i in range(1, len(A)):
7          for j in range(0, len(A)-1):
8              cnt+=1
9              if(A[j+1]<A[j]):
10                 aux=A[j]
11                 A[j]=A[j+1]
12                 A[j+1]=aux
13                 #print(A)
14             return A
15
16  #programa principal
17  print("ARREGLO DESORDENADO:")
18  A=[6,5,3,1,8,7,2,4]
19  print(A)
20  print("ARREGLO ORDENADO: \n", burbuja(A))
21  input("presione enter para continuar")
```



The screenshot shows a Windows command prompt window titled "C:\Windows\py.exe". The output of the Python script is displayed in white text on a black background. It shows the initial unsorted array [6, 5, 3, 1, 8, 7, 2, 4] and the final sorted array [1, 2, 3, 4, 5, 6, 7, 8]. The prompt "presione enter para continuar" is visible at the bottom.

```
C:\Windows\py.exe
ARREGLO DESORDENADO:
[6, 5, 3, 1, 8, 7, 2, 4]
ARREGLO ORDENADO:
[1, 2, 3, 4, 5, 6, 7, 8]
presione enter para continuar
```

Ordenamiento por Selección.

1. Descripción.

Este algoritmo también es sencillo. Consiste en lo siguiente:

- Buscas el elemento más pequeño de la lista.
- Lo intercambias con el elemento ubicado en la primera posición de la lista.
- Buscas el segundo elemento más pequeño de la lista.
- Lo intercambias con el elemento que ocupa la segunda posición en la lista.
- Repites este proceso hasta que hayas ordenado toda la lista.

2. Análisis del algoritmo.

- Requerimientos de Memoria: Al igual que el ordenamiento burbuja, este algoritmo sólo necesita una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: El ciclo externo se ejecuta n veces para una lista de n elementos. Cada búsqueda requiere comparar todos los elementos no clasificados. Luego la complejidad es $O(n^2)$. Este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad promedio es también $O(n^2)$.

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.
- Realiza pocos intercambios.
- Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

- Lento.

- Realiza numerosas comparaciones.

Este es un algoritmo lento. No obstante, ya que sólo realiza un intercambio en cada ejecución del ciclo externo, puede ser una buena opción para listas con registros grandes y claves pequeñas. La razón es que es mucho más lento dibujar las barras que comparar sus largos (el desplazamiento es más costoso que la comparación), por lo que en este caso especial puede vencer a algoritmos como Quicksort.

```
1  #Lealie Jaressy Morales Ortiz 1661971
2  #Matematicas Computacionales
3
4  >>> cnt=0
5  >>> def selection(arr):
6      global cnt
7      for i in range(0,len(arr)-1):
8          val=i
9          for j in range (i+1,len(arr)):
10             cnt=cnt+1
11             if arr[j]<arr[val]:
12                 val=j
13             if val!=i:
14                 aux=arr[i]
15                 arr[i]=arr[val]
16                 arr[val]=aux
17         return arr
18
19  >>> A=[3,1,89,7]
20  >>> selection(A)
21  [1, 3, 7, 89]
22  >>>
```

Ordenamiento por Inserción.

1. Descripción.

Este algoritmo también es bastante sencillo. ¿Has jugado cartas? ¿Cómo las vas ordenando cuando las recibes? Yo lo hago de esta manera: tomo la primera y la coloco en mi mano. Luego tomo la segunda y la comparo con la que tengo: si es mayor, la pongo a la derecha, y si es menor a la izquierda (también me fijo en el color, pero omitiré esa parte para concentrarme en la idea principal). Después tomo la tercera y la comparo con las que tengo en la mano, desplazándola hasta que quede en su posición final. Continúo haciendo esto, *insertando* cada carta en la posición que le corresponde, hasta que las tengo todas en orden. ¿Lo haces así tu también? Bueno, pues si es así entonces comprenderás fácilmente este algoritmo, porque es el mismo concepto.

Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos, así como así o se perderá un elemento. Lo que hacemos es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.

2. Análisis del algoritmo.

- Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto, es *estable*.
- Requerimientos de Memoria: Una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: Para una lista de n elementos el ciclo externo se ejecuta $n-1$ veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad $O(n^2)$.

Ventajas:

- Fácil implementación.
- Requerimientos mínimos de memoria.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Este también es un algoritmo lento, pero puede ser de utilidad para listas que están ordenadas o semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

```
1  cnt=0
2  #Leslie Jaressy Morales Ortiz.
3  #Insercion
4  def orden_por_insercion(array):
5      global cnt
6      for indice in range (1,len(array)):
7          valor=array[indice]#valor es el elemento que estamos comparando
8          i=indice-1      #i es el valor anterior al elemento que estamos comparando
9          while i>=0:
10             cnt+=1
11             if valor<array[i]:#comparamos valor con el elemento anterior
12                 array[i+1]=array[i]#intercambiamos los valores
13                 array[i]=valor
14                 i-=1#Decrementamos en 1 el valor de i
15             else:
16                 break
17         return array
18
19  >>> A=[56,5,48,1]
20  >>> orden_por_insercion(A)
21  [1, 5, 48, 56]
22  >>>
```

Ordenamiento por Quicksort

1. Descripción.

Esta es probablemente la técnica más rápida conocida. Fue desarrollada por C.A.R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos). El algoritmo fundamental es el siguiente:

- Eliges un elemento de la lista. Puede ser cualquiera (en Optimizando veremos una forma más efectiva). Lo llamaremos **elemento de división**.
- Buscas la posición que le corresponde en la lista ordenada (explicado más abajo).
- Acomodas los elementos de la lista a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores (explicado más abajo también). En este momento el elemento de división separa la lista en dos sablistas (de ahí su nombre).
- Realizas esto de forma recursiva para cada sablista mientras éstas tengan un largo mayor que 1. Una vez terminado este proceso todos los elementos estarán ordenados.

Una idea preliminar para ubicar el elemento de división en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo. Se utilizan dos índices: *i*, al que llamaremos contador por la izquierda, y *j*, al que llamaremos contador por la derecha. El algoritmo es éste:

- Recorres la lista simultáneamente con *i* y *j*: por la izquierda con *i* (desde el primer elemento), y por la derecha con *j* (desde el último elemento).

- Cuando `lista[i]` sea mayor que el elemento de división y `lista[j]` sea menor los intercambias.
- Repites esto hasta que se crucen los índices.
- El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

2. Optimizando.

Sólo voy a mencionar algunas optimizaciones que pueden mejorar bastante el rendimiento de quicksort:

- Hacer una versión iterativa: Para ello se utiliza una pila en que se van guardando los límites superior e inferior de cada sublista.
- No clasificar todas las sublistas: Cuando el largo de las sublistas va disminuyendo, el proceso se va *encareciendo*. Para solucionarlo sólo se clasifican las listas que tengan un largo menor que **n**. Al terminar la clasificación se llama a otro algoritmo de ordenamiento que termine la labor. El indicado es uno que se comporte bien con listas casi ordenadas, como el ordenamiento por inserción, por ejemplo. La elección de **n** depende de varios factores, pero un valor entre 10 y 25 es adecuado.
- Elección del elemento de división: Se elige desde un conjunto de tres elementos: `lista[inferior]`, `lista[mitad]` y `lista[superior]`. El elemento elegido es el que tenga el valor medio según el criterio de comparación. Esto evita el comportamiento degenerado cuando la lista está prácticamente ordenada.

3. Análisis del algoritmo.

- Estabilidad: No es *estable*.

- Requerimientos de Memoria: No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila.
- Tiempo de Ejecución:
 - Caso promedio. La complejidad para dividir una lista de n es $O(n)$. Cada sublista genera en promedio dos sublistas más de largo $n/2$. Por lo tanto, la complejidad se define en forma recurrente como:

$$f(1) = 1$$

$$f(n) = n + 2 f(n/2)$$
 La forma cerrada de esta expresión es:

$$f(n) = n \log_2 n$$
 Es decir, la complejidad es **$O(n \log_2 n)$** .
 - El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a $O(n^2)$. Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.

Ventajas:

- Muy rápido
- No requiere memoria adicional.

Desventajas:

- Implementación un poco más complicada.
- Recursividad (utiliza muchos recursos).
- Mucha diferencia entre el peor y el mejor caso.

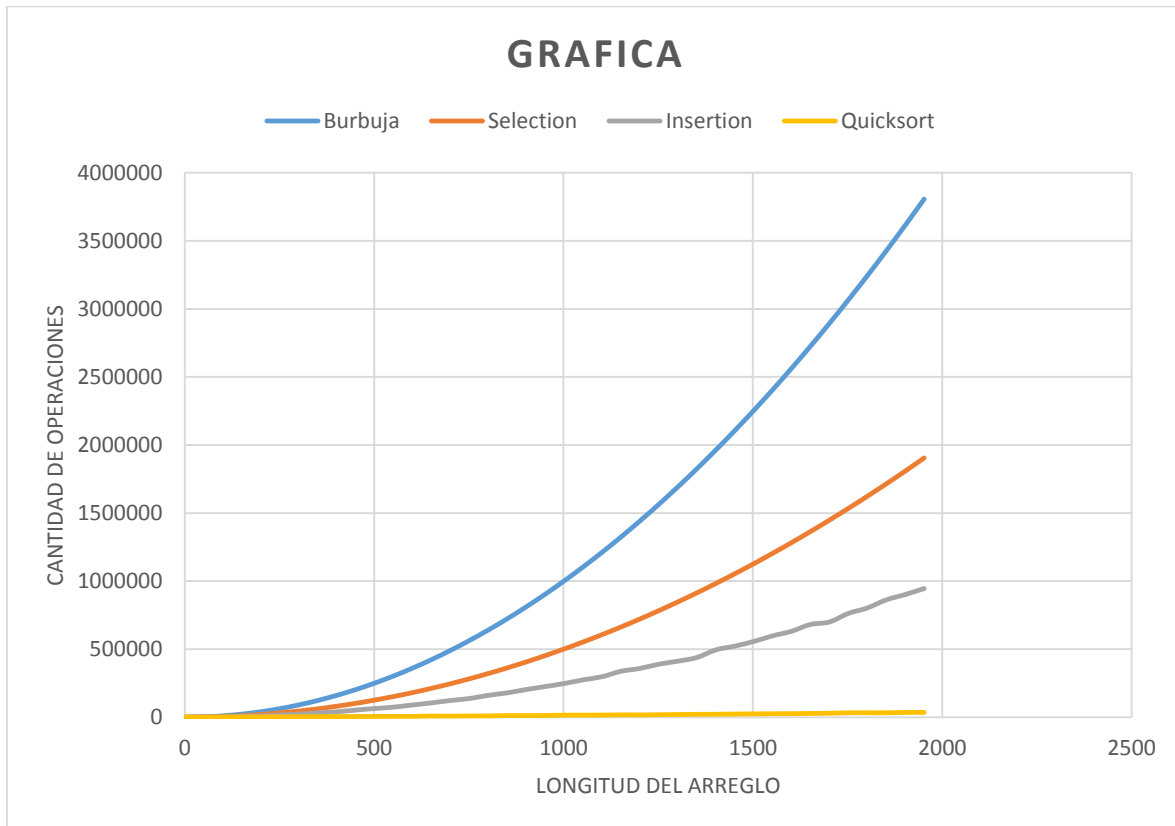
La mayoría de los problemas de rendimiento se pueden solucionar con las optimizaciones mencionadas arriba (al costo de complicar mucho más la implementación). Este es un algoritmo que puedes utilizar en la vida real.

```
1  import random
2  cnt = 0
3
4  def quicksort(arr):
5      global cnt
6      if len(arr) < 2:
7          return arr
8      p = arr.pop(0)
9      menores, mayores = [], []
10     for e in arr:
11         cnt +=1
12         if e <= p:
13             menores.append(e)
14         else:
15             mayores.append(e)
16     return quicksort(menores) + [p] + quicksort(mayores)
17
18 def rndar(long):
19     arr = []
20     for i in range(long):
21         arr.append(random.randint(0,long))
22     return arr
23
24
25
26  l = 10
27
28
29
30  while l <= 10:
31     for replica in range(10):
32         ori = rndar(l)
33         arr = quicksort(ori)
34         print( l, cnt, arr, ori)
35         cnt = 0
36     l*=2
```

Tabla

Longitud	Burbuja	Selection	Insertion	Quicksort
2	1	1	0	1
52	2601	1326	702	314
102	10201	5151	2511	612
152	22801	11476	6016	1106
202	40401	20301	9544	1615
252	63001	31626	15432	2104
302	90601	45451	21954	2581
352	123201	61776	31479	3281
402	160801	80601	39367	4067
452	203401	101926	52435	4489
502	251001	125751	63374	4935
552	303601	152076	74024	5726
602	361201	180901	89485	5912
652	423801	212226	105126	7728
702	491401	246051	122452	7436
752	564001	282376	137063	8662
802	641601	321201	160746	9357
852	724201	362526	178677	11433
902	811801	406351	203124	11414
952	904401	452676	224883	12059
1002	1002001	501501	247217	14188
1052	1104601	552826	274081	14176
1102	1212201	606651	297484	14787
1152	1324801	662976	337123	16722
1202	1442401	721801	357977	16469
1252	1565001	783126	389257	17412
1302	1692601	846951	411014	18834
1352	1825201	913276	438137	20100
1402	1962801	982101	494099	20748
1452	2105401	1053426	521608	22245
1502	2253001	1127251	555807	23196
1552	2405601	1203576	597110	24136
1602	2563201	1282401	630743	24896
1652	2725801	1363726	680940	26398
1702	2893401	1447551	698758	27887
1752	3066001	1533876	761924	30991
1802	3243601	1622701	801813	31216
1852	3426201	1714026	861139	31036
1902	3613801	1807851	900369	33703
1952	3806401	1904176	944523	33968

Grafica.



Conclusión General.

Desde mi punto de vista considero que cada uno de estos algoritmos de ordenación es útil en distintos tiempos pero cabe destacar en la gráfica donde es más evidente el grado de complejidad de estos , en lo particular me pareció que el burbuja fue uno de los más sencillos de poner en práctica , a decir verdad tengo que confesar que la dificultad para la realización de los próximos 3 fue creciendo hasta llegar a quicksort ,el cual fue el tema que nos tocó como equipo , al principio fue fácil de comprender el algoritmo pero al momento de trabajar con Python las dificultades surgieron nuevamente .

Así mismo tengo que resaltar que fue muy interesante trabajar con un nuevo programador, en realidad tengo toda la iniciativa de conocer sobre cosas nuevas