# Comparación de algoritmos de ordenamiento

Luis Carlos Aguilar Saucedo

9 de abril de 2018

## 1. Introducción

Los algoritmos de ordenamiento son todos los algoritmos que, por medio de sus instrucciones, consiguen un orden específico de los elementos de arreglos. Dichos elementos son regularmente numéricos. Las maneras en las que estos algoritmos cumplen su función son diversas, tienen enfoques distintos, ya sea la agrupación de elementos o su manejo individual. Uno de los criterios principales por los que se juzga un algoritmo de ordenamiento es su eficiencia, es decir, la capacidad de llegar al resultado deseado en el menor número de pasos, lo que a su vez se traduce en menor tiempo de ejecución.

## 2. Algoritmos de ordenamiento

En este trabajo se comparan los algoritmos de ordenamiento conocidos como Bubble, Selection, Insertion y Quicksort, los cuales se presentan a continuación. Todos estos algoritmos reordenan los elementos de arreglos numéricos.

Algoritmo Bubble

El algoritmo Bubble, Burbuja en español, es un algoritmo de ordenamiento que consigue un orden ascendente o descendente de los elementos de un arreglo mediante una comparación por la que determina cuál es el elemento mayor o menor del arreglo, para después colocarlo en el extremo deseado del mismo o de un nuevo arreglo. El proceso se repite para todos los elementos restantes, colocándolos antes del último elemento reordenado hasta que los elementos se agotan y todos han sido acomodados. La siguiente es una representación del funcionamiento del algoritmo Burbuja.

Paso 0.6342182. Arreglo original.

Comenzando con el primer elemento, compara cada uno, colocando el mayor a la derecha. Esto quiere decir que el primer elemento, el 6, que es mayor que los siguientes cuatro, se trazladará hasta esa posición(pasos 1 - 4). El 8, que es

mayor que el 6, será el siguiente en trasladarse (pasos 5 - 6). Como ya no hay elementos mayores que el 8, éste llegará hasta la posición final.

Paso 1. 3642182. Con $6 > 3$
Paso 2. 3462182. Con $6 > 4$
Paso 3. 3426182. $6 > 2$
Paso 4. 3421682. $6 > 1$
Paso 5. 3421682. $6 < 8$
Paso 6. 3421628. $8 > 2$

El algoritmo repetirá esta operación con todos los elementos del arreglo excepto el último, que ya es el mayor. Por ello, el nuevo elemento mayor quedará justo antes del que ya ocupa el último lugar.

Paso 7. 3421628. $3 < 4$
Paso 8. 3241628. $4 > 2$
Paso 9. 3214628. $4 > 1$
Paso 10. 3214628. $4 < 6$
Paso 11. 3214268. $6 > 2$

De esta manera, al repetir la comparación de todos los elementos anteriores a cada elemento reacomodado, el resultado será un arreglo cuyos elementos estén reordenados de menor a mayor. El código de este algoritmo y los demás se mostrará en la siguiente sección.

Algoritmo Selection

Este algoritmo funciona de manera similar al anterior. En lugar de comparar cada elemento con el que le sigue, considera al primero como el mayor y lo compara con todos hasta hallar uno mayor, y luego ése hasta hallar otro mayor, repitiendo este proceso hasta terminar los elementos. El que resultara mayor en la comparación ocupará el último lugar del arreglo.

Paso 0. 6342182. Arreglo original. El primer elemento, el número 6, se considera el mayor.
Paso 1. 6342182. $6 > 3$
Paso 2. 6342182. $6 > 4$
Paso 3. 6342182. $6 > 2$
Paso 4. 6342182. $6 > 1$
Paso 5. 6342182. $6 < 8$. El 8 se elige temporalmente como el elemento mayor.
Paso 6. 6342182. $8 > 2$
Paso 7. 6342128. El último elemento y el elemento determinado mayor se intercambian.

En cada siguiente comparación, el elemento que resulte mayor ocupará el lugar anterior al último. El proceso se repetirá, cada elemento mayor ocupando un lugar anterior.

Algoritmo Insertion

En el algoritmo Insertion se toma cada elemento de la lista a partir del segundo. El elemento seleccionado se compara con los anteriores y se reubica antes de los elementos que sean mayores que éste. Para conseguirlo, el algoritmo hace que los elementos anteriores al elegido ocupen un lugar posterior al que ocupaban originalmente. Solamente se desplazan los elementos mayores al elegido, y el elemento elegido ocupa el lugar del último desplazado.

Paso 0. 6342182. El primer elemento elegido será el 3.
Paso 1. 3642182. $6 > 3$. El siguiente elemento elegido es el 4.
Paso 2. 3462182. $6 > 4$ y $3 < 4$. Por lo tanto, solamente se desplaza el 6 y el 4 ocupa su lugar. El 2 es el siguiente elemento elegido.
Paso 3. 3426182. $6 > 2$
Paso 4. 3246182. $4 > 2$
Paso 5. 2346182. $3 > 2$. El 1 es el siguiente elemento elegido.

El algoritmo continúa hasta que todos los elementos han sido elementos elegidos.

Algoritmo Quicksort

Para ordenar los elementos de un arreglo, este algoritmo toma uno de los elementos del mismo, al que se le llama pivote. Después compara los demás elementos con el pivote y los coloca, si son mayores, en un grupo de elementos mayores; si son menores, en un grupo de menores. En cada uno de esos grupos se elige a su vez un pivote sobre cuya comparación se depositan los demás en subgrupos, a los cuales se les realizará la misma operación hasta que todos los subgrupos hayan sido separados. Después, los grupos son reunidos, colocando los elementos menores antes de los pivotes y los mayores, detrás.

Paso 0. 6342182. Se elige un pivote, que puede ser cualquier elemento. En este caso se elige el primer 2.
Paso 1. El 1 constituye los elementos menores al pivote. Ambos 2 forman un grupo de elementos iguales al pivote. 6348 constituye el grupo de elementos mayores. Se eligen pivotes en los grupos que tengan más de un elemento. 2 es el pivote del grupo de elementos iguales al pivote anterior. 4 es el pivote del grupo de elementos mayores que el pivote anterior.
Paso 2. En el grupo cuyo pivote actual es 2, los únicos elementos son iguales al pivote, por lo que no se forman grupos de mayores y menores. En el grupo

cuyo pivote actual es 4,se forma el grupo de menores con 3, y el de mayores con 68. Sólo en este último grupo se elige el 8 como nuevo pivote.

Paso 3. En el último grupo, 6 entra en el grupo de menores que 8. En este punto, todos los grupos tienen un solo elemento.

Paso 4. Se une el grupo que forma 68. Este grupo se une al que era su pivote, el 4 y a ellos los antecede el 3, que era el grupo de menores que 4. Así se forma 3468. A éstos se une el grupo con ambos 2, y a su vez les antecede el grupo con 1. El resultado es 1223468.

# 3. Código de los algoritmos

Los siguientes son los códigos de los algoritmos arriba descritos. Todos los algoritmos fueron escritos en python. Los comentarios se agregan con barras dobles (//).

Bubble.

```
>def bubbleSort(alist):
>>global contador
// La variable contador aparece en todos los códigos de los
//algoritmos para llevar la cuenta de las operaciones que
//realizan durante el ordenamiento.
>>for passnum in range(len(alist)-1,0,-1):
//El primer valor de este rango equivale a la última posición del arreglo.
//La instrucción para los valores de este rango es disminuir hasta 0.
//passnum define el lugar que tomará el elemento mayor de cada
//comparación antes de cambiar de valor.
>>>for i in range(passnum):
//i toma todos los valores de passnum.
>>>>contador += 1
//El contador sube 1 cada vez que se hace una comparación.
>>>>if alist[i]¿alist[i+1]:
//"si un elemento es mayor que el que le sigue..."
//se les intercambia.
>>>>>temp = alist[i]
>>>>>alist[i] = alist[i+1]
>>>>>alist[i+1] = temp
//Después de este intercambio hasta que el elemento mayor ocupe
//el último lugar, passnum reduce 1 su valor, por lo que el siguiente
//elemento mayor ocupará un lugar anterior al que ya fue ocupado.
>>return contador
//Con este retorno, la función entrega el número de pasos que le toma
//al algoritmo ordenar el arreglo.
```

Selection

```
>def selectionSort(alist):
>>global contador
>>for fillslot in range(len(alist)-1,0,-1):
//El primer valor de este rango equivale a la última posición del arreglo.
//La instrucción para los valores de este rango es disminuir hasta 0.
>>>positionOfMax=0
//Con esta definición se considerará después al primer elemento
//al inicio de cada comparación, como si fuera el mayor.
>>>for location in range(1,fillslot+1):
//El primer valor es 1 porque se comparará el elemento 1 con
//el elemento 0, dado por positionOfMax.
//Se considera hasta el elemento siguiente dentro del rango de fillslot.
//De esta manera, cuando el elemento mayor ya ocupe el último
//lugar y fillslot disminuya una unidad, ese elemento colocado
//al final ya no será movido.
>>>>contador +=1
//El contador sube 1 cada vez que se hace una comparación.
>>>>if alist[location]¿alist[positionOfMax]:
//"si un elemento es mayor a su elemento anterior..."
//se considerará que la posición del mayor es la posición
//determinada por el valor de location.
>>>>>positionOfMax = location
//El elemento mayor ocupará el último lugar, asignado por
//fillslot antes de reducir su valor para iniciar una nueva comparación.
//El elemento que ocupaba el último ligar se coloca en el lugar que
//ocupaba el mayor.
>>>temp = alist[fillslot]
>>>alist[fillslot] = alist[positionOfMax]
>>>alist[positionOfMax] = temp
>>return contador
```

Insertion

```
>def insertionSort(alist):
>>global contador
>>for index in range(1,len(alist)):
>>>currentvalue = alist[index]
//Se guardará el valor del elemento index del arreglo, o sea que el
//primer valorguardado será el del segundo elemento.
>>>position = index
//La posición y el index valen lo mismo, pero la posición cambiará,
//el index no.
>>>contador += 1
```

>>>while position¿0 and alist[position-1]¿currentvalue:
//Deben cumplirse ambas condiciones para que haya cambios de lugar.
//Si hay cambios de lugar hasta el comienzo del arreglo, la posición
//valdrá 0 y el ciclo se romperá.
//Si un elemento no es mayor que el valor guardado, el ciclo
//se romperá. La posición no valdrá cero, entonces ése será el
//lugar donde se acomode el elemento equivalente al valor
//guardado, menor que los demás que desplazó, pero
//mayor que el elemento con el que se rompió el ciclo.
>>>>contador += 1
//De la siguiente manera se desplazan los elementos:
//cada elemento mayor que el valor ocupa el lugar del
//elemento que lo sigue.
>>>>alist[position]=alist[position-1]
>>>>position = position-1
//Y enseguida se hace que el elemento que sigue al que fue
//menor que el valor guardado, tome este valor.
>>>alist[position]=currentvalue
>>return contador

Quicksort

>def quickSort (alist):
>>global contador
>>if len(alist)¡ 2:
//Si sólo hay un elemento en el arreglo, no se le modifica.
>>>return alist
//Se crean los grupos en los que se distribuirán los elementos del
//arreglo.
>>menores = []
>>iguales = []
>>mayores = []
>>pivote = alist[round(len(alist)/2)]
//Se determina el pivote. En este caso, un pivote intermedio.
>>for x in alist:
>>>contador += 1
//Con esta serie de ifs, se reparten los elementos.
>>>if x ¿ pivote:
>>>>mayores.append(x)
>>>if x ¡ pivote:
>>>>menores.append(x)
>>>if x == pivote:
>>>>iguales.append(x)
>>return quickSort(menores) + iguales + quickSort(mayores)
//En el retorno de la función, ésta se reitera en los grupos que se

//crearon. Cuando todos los grupos tienen un solo elemento,
//ya no los modifica y los reúne de menor a mayor.
//Adicionalmente, se observa que el retorno de esta función es un
//arreglo, no un contador. El contador sí se modifica durante la
//ejecución del algoritmo, pero es llamado desde fuera de la función
//para conocerlo.

# 4. Comparación de los algoritmos

La siguiente es una prueba para comparar la eficiencia de los algoritmos. Se obtienen muestras significativas de los contadores señalados por los algoritmos. Cada algoritmo es sometido treinta veces a nueve ordenamientos de arreglos aleatorios de distintas longitudes. Así, se comenzará por ordenar treinta arreglos aleatorios con cuatro elementos. Después otros treinta con el doble de elementos que los anteriores, hasta completar treinta veces los nueve ordenamientos. Las cantidad de elementos por ordenamiento es, pues, la siguiente:

Primero. 4 elementos.
Segundo. 8 elementos.
Tercero. 16 elementos.
Cuarto. 32 elementos.
Quinto. 64 elementos.
Sexto. 128 elementos.
Séptimo. 256 elementos.
Octavo. 512 elementos.
Noveno. 1024 elementos.

El código para llamar a los algoritmos para esta prueba es el siguiente. Se incluyen comentarios.

>l = 4
//Éste es el número de elementos inicial.
>print("L","B","S",Ï", "Q")
//Estas letras establecen el encabezado de las columnas de datos.
//L por length, B por Bubble, S por Selection, I por Insertion
//y Q por Quicksort.
>while l ¡ 1100:
//l incrementará durante la reiteración. Como el número mayor de
//elementos que habrá será 1024, y la razón con la que incrementará
//será *= 2, es decir, cada vez el doble, 1100 es límite suficiente
//para detener la reiteración.
>>for replica in range(30):
//Se pedirá treinta veces un arreglo con una longitud determinada.
>>> arr = aRand(l)

//El arreglo medirá lo que l indique.
>>>a1, a2, a3, a4 = copy.deepcopy(arr), copy.deepcopy(arr), copy.deepcopy(arr), copy.deepcopy(arr)
//El arreglo generado se copia una vez para cada algoritmo que lo
//ordenará, con nombres distintos para cada copia.
//Enseguida se crean variables que se hacen equivaler a
//los retornos de las funciones, es decir, a los contadores que
//ellas regresan.
>>>bc = bubbleSort(a1)
>>>sc = selectionSort(a2)
>>>ic = insertionSort(a3)
//Para Quicksort es necesario declarar previamente un contador.
>>>qc = 0
>>>r4 = quickSort(a4)
//Se imprime en una fila la cantidad de elementos en el arreglo,
//seguida de las cuentas de cada algoritmo. Esto ocurre treinta veces.
>>>print(l, bc, sc, ic, qc)
//Enseguida cambia el valor de l y se repite la invocación de las funciones.
//La invocación se detiene cuando l vale 2048.
//En total, los algoritmos realizaron 270 ordenaciones cada uno.
>>l *= 2

Posterior a la generación de datos, se crea una gráfica que muestra la tendencia de los algoritmos durante los ordenamientos, cada uno con respecto de los otros.

El código para esta gráfica se escribió en R y es el siguiente.

```
datos = read.csv("Data.txt", sep = " ")

png(filename= "AlgoritmosDeOrdenamiento.png")
boxplot(log(datos[,-1]), main = "Comparación de algoritmos de ordenamiento")
graphics.off()
```

El archivo Data.txt contiene los datos de la prueba. Con png se determina que la grafica a ser generada debe ser escrita en ese formato. Finalmente, en boxplot se determina que el tipo de gráfica será de caja y bigotes. En esta instrucción se determina, con datos(), que y serán todas las filas de la tabla, mientras que x, todas las columnas menos la primera, la que contiene las longitudes de los arreglos.

# 5. Resultados

En primer lugar se toma una muestra de los datos directamente arrojados por la prueba. Se recuerda que la primera columna dice cuántos elementos hay en el arreglo, las siguientes cuatro son las cuentas para Bubble, Selection, Insertion y Quicksort, respectivamente. La lista entera de datos incluye treinta filas para cada longitud. Aquí se toman indistintamente las primeras tres filas de cada longitud.

L B S I Q
4 6 6 6 7
4 6 6 4 6
4 6 6 7 6
8 28 28 21 23
8 28 28 17 21
8 28 28 18 29
16 120 120 63 54
16 120 120 77 58
16 120 120 50 56
32 496 496 272 133
32 496 496 243 144
32 496 496 259 128
64 2016 2016 1099 355
64 2016 2016 927 367
64 2016 2016 1031 475
128 8128 8128 3737 909
128 8128 8128 4274 776
128 8128 8128 4185 775
256 32640 32640 16039 1744
256 32640 32640 15517 1925
256 32640 32640 15613 1885
512 130816 130816 64288 4047
512 130816 130816 64364 3729
512 130816 130816 62704 3747
1024 523776 523776 259379 6795
1024 523776 523776 266949 7053
1024 523776 523776 261274 7006

La gráfica en la siguiente página representa las tendencias de los algoritmos en su eficiencia según el número de pasos en todos los 270 ordenamientos.

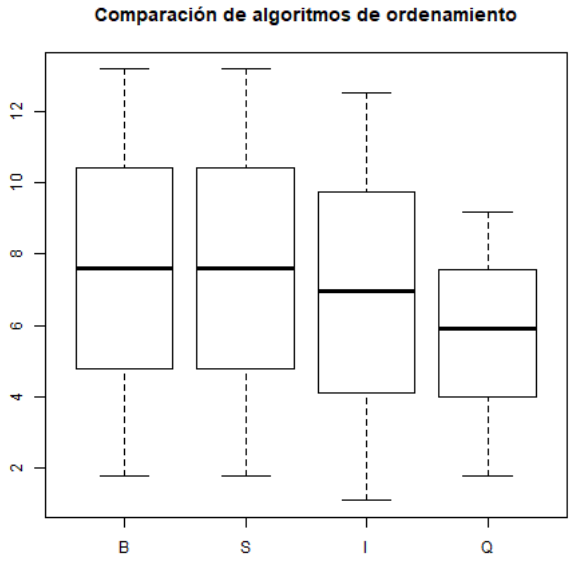


Figura 1: Como se mencionó en el código de la gráfica, en x se muestran las letras que representan los algoritmos de ordenamiento. B: Bubble, S: Selection, I: Insertion, Q: Quicksort.

# 6. Análisis de resultados y comentarios finales.

Uno de los primeros detalles destacables es que los algoritmos Bubble y Selection hacen el mismo número de pasos para sus ordenamientos. Teniendo en cuenta los códigos arriba descritos, es posible detectar por qué coinciden en sus cuentas. Ambos hacen recorridos por todo el arreglo para cada comparación, indistintamente del orden en el que aparezcan los elementos los elementos. Tanto la lista como la gráfica muestran que Bubble y Selection son los algoritmos que más pasos requieren para ordenar los arreglos.

Insertion y Quicksort muestran comportamientos distintos. La lista revela que cuando se trata de arreglos cortos, estos algoritmos no tienen cuentas muy distintas, e incluso Insertion llega a ser más eficiente que Quicksort en arreglos de longitud 16 y menores. Sin embargo, conforme incrementa el número de elementos por arreglo, Quicksort demuestra ser más eficiente.

Quicksort, en efecto, no supera la cantidad de diez mil pasos en ningún ordenamiento de la prueba, mientras que los otros tres superan esta cantidad ya con arreglos de 256 elementos. Bubble y Selection superan los quinientos mil pasos para arreglos de 1024 elementos, e Insertion supera los doscientos cincuenta mil.

La lista y gráfica muestran que la tendencia es que, conforme incrementa el

número de elementos en un arreglo, Quicksort incrementa la cuenta de pasos para ordenar el arreglo en una medida menor que la de los otros tres algortimos.