

Matemáticas computacionales

Segundo Reporte de algoritmo de ordenamiento (Mejorado)

Alumno: Isaac Emanuel Segovia Olay

M. 1748957

Maestro: Lic. José Anastacio Hernández Saldaña

18-septiembre-2017

En este segundo reporte se describirán brevemente los algoritmos que ya hemos visto en clases que son los algoritmos de ordenación que usamos en Python, la explicación de ellos y agregando los algoritmos que hicimos en clase al reporte anterior, más una conclusión de cada uno de ellos mencionando las ventajas y desventajas que tiene cada uno.

Además, incluirá el reporte gráficas para valorar los códigos de los algoritmos de ordenación y medir su desempeño para poder comparar las cuatro que hemos visto hasta ahora.

Inserción

En clases, la idea que tuvimos del ordenamiento por inserción es tener dos arreglos en el cual uno está ordenado y el otro no. Así comenzamos con el algoritmo, tomamos el primer elemento el cual está ordenado e iteramos sobre el arreglo desordenado tomando un elemento a la vez y cuando tomemos ese elemento lo insertamos en un arreglo ordenado, así podremos tener el elemento que está desordenado en el arreglo ordenado de tal modo que ese elemento esté en su posición que deba de ir, de este modo el algoritmo acaba cuando el arreglo desordenado esté vacío sin ningún elemento

Insertion.py - C:\Users\isaac\Desktop\Mat Comp\Insertion.py (3.6.2)

File Edit Format Run Options Window Help

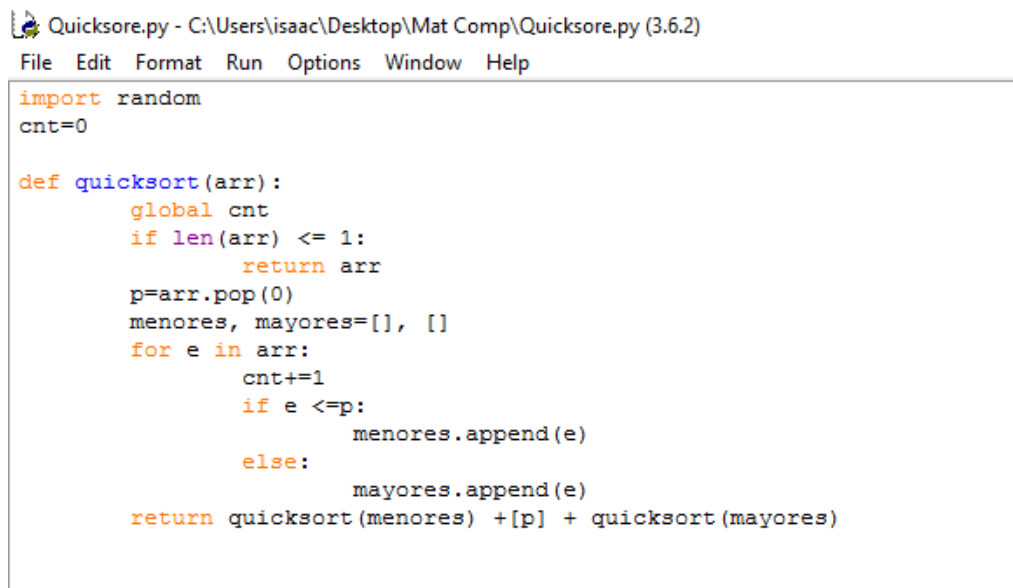
```
#Insercion
cnt=0
def orden_por_insercion(array):
    global cnt
    for indice in range(1,len(array)):
        valor=array[indice]
        i=indice-1
        while i>=0:
            cnt+=1
            if valor<array[i]:
                array[i+1]=array[i]
                array[i]=valor
                i-=1
            else:
                break
    return array
```

En este algoritmo nos podemos dar cuenta que es muy eficiente ya que en el mejor de los casos no intercambia nada nunca así solo tardaría una vez en recorrer el arreglo. Y en el peor de los casos sería de n^2 lo cual es cuadrático en función del arreglo.

También podemos realizar el proceso sobre el mismo arreglo de ordenamiento, pero como tiene que iterar en todo el arreglo para tomar los elementos desordenados y también en el arreglo ordenado para poner elementos, entonces la complejidad si es cuadrática.

Quicksort

Cuando mis compañero me explicaron sobre quicksort es que es el algoritmo de ordenación más rápido conocido, su tiempo de ejecución promedio es $O(n \log(n))$, siendo en el peor de los casos $O(n^2)$, caso altamente improbable. El hecho de que sea más rápido que otros algoritmos de ordenación con tiempo promedio de $O(n \log(n))$ viene dado por que QuickSort realiza menos operaciones ya que el método utilizado es el de partición. Una explicación de este algoritmo sacado de internet es que se elige un elemento v de la lista L de elementos al que se le llama pivote. Se particiona la lista L en tres listas: $L1$ - que contiene todos los elementos de L menos v que sean menores o iguales que v . $L2$ - que contiene a v . $L3$ - que contiene todos los elementos de L menos v que sean mayores o iguales que v . Se aplica la recursión sobre $L1$ y $L3$. Se unen todas las soluciones que darán forma final a la lista L finalmente ordenada. Como $L1$ y $L3$ están ya ordenados, lo único que tenemos que hacer es concatenar $L1$, $L2$ y $L3$.



```
Quicksore.py - C:\Users\isaac\Desktop\Mat Comp\Quicksore.py (3.6.2)
File Edit Format Run Options Window Help


import random
cnt=0

def quicksort(arr):
    global cnt
    if len(arr) <= 1:
        return arr
    p=arr.pop(0)
    menores, mayores=[], []
    for e in arr:
        cnt+=1
        if e <=p:
            menores.append(e)
        else:
            mayores.append(e)
    return quicksort(menores) +[p] + quicksort(mayores)
```

Este es el algoritmo en el que es más óptimo, es muy sencillo de programar aun cuando tiene como gran ventaja en tiene mejor promedio de tiempo que los demás. A pesar que la complejidad del algoritmo sea de $O(n \log(n))$, en el peor de los caso puede llegar ser de n^2 . Cuando los arreglos están en un cierto punto ordenados y con muchos valores repetidos su disminuye su tiempo.

Bubble

Este algoritmo me tocó junto con mi equipo en explicar a la clase y lo que sé ahora es que funciona comparando en el vector dos elementos adyacentes, si el segundo elemento es menor al primero entonces son intercambiados, en caso contrario se evalúa el segundo elemento y el siguiente elemento adyacente a su izquierda. Ya cuando esté terminada la primera iteración se vuelve a realizar el mismo procedimiento sin embargo en cada iteración hecha, un elemento del vector, de derecha a izquierda, ya no es evaluado ya que contiene el número mayor de cada iteración. El algoritmo terminará una vez que el número de iteraciones sea igual al número de elementos que contenga el vector de números.

 Burbuja.py - C:\Users\isaac\Desktop\Mat Comp\Burbuja.py (3.6.2)

File Edit Format Run Options Window Help

```
def burbuja(A):  
    for i in range(1, len(A)):  
        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  
  
    print(A)
```

La complejidad de este algoritmo es de n^2 en el peor de los casos, el cual es cuando el arreglo está completamente invertido, se tendrá que recorrer en el arreglo $n-1$ veces y deberá de hacer una iteración n veces.

En este algoritmo no tiene un buen desempeño, pero su interpretación es muy fácil de entender y no requiere espacio adicional y eso es de gran ayuda cuando hay poca memoria de espacio.

Selection

Por último, en este algoritmo que es selection lo que hace es que la función principal, es la encargada de recorrer la lista, ubicando el mayor elemento al final del segmento y luego reduciendo el segmento a analizar. También busqué más información y encontré que busca el mayor de todos los elementos de la lista. Poner el mayor al final (intercambiar el que está en la última posición de la lista con el mayor encontrado). Buscar el mayor de todos los elementos del segmento de la lista entre la primera y la anteúltima posición. Poner el mayor al final del segmento (intercambiar el que está en la última posición del segmento, o sea anteúltima posición de la lista, con el mayor encontrado). Se termina cuando queda un único elemento sin tratar: el que está en la primera posición de la lista, y que es el menor de todos porque todos los mayores fueron reubicados.

```
Selection.py - C:\Users\isaac\Desktop\Mat Comp\Selection.py (3.6.2)
File Edit Format Run Options Window Help

#Selection
def selection(arr):
    for i in range(0, len(arr)-1):
        val=i
        for j in range(i+1, len(arr)):
            if arr[j]<arr[val]:
                val=j
        if val!=i:
            aux=arr[i]
            arr[i]=arr[val]
            arr[val]=aux
    return arr
```

Su cantidad de operaciones en este algoritmo será cuadrática con respecto al arreglo, igual que el algoritmo de inserción, puede ordenar el arreglo por cada iteración que haga y no usa memoria adicional.

Puede que la complejidad del algoritmo no sea tan complicada de notar, ya que repite el proceso todas las veces que restan de elementos en el arreglo, así que podemos decir que no es muy bueno en cuanto su desempeño si lo comparamos con los demás.

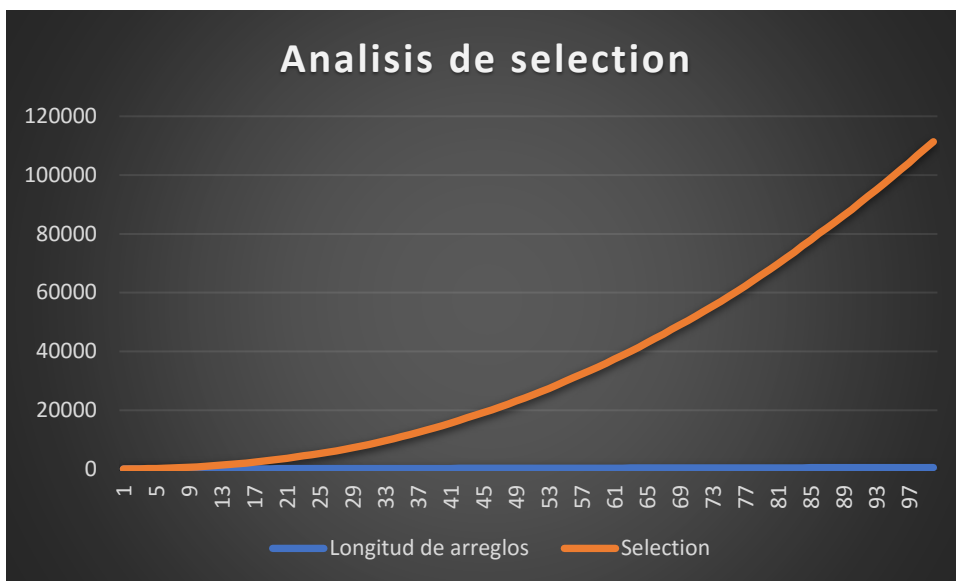
Gráficas (Análisis de complejidad)

Ahora que ya entendemos los algoritmos que hemos usado en Python, Ahora probaremos los algoritmos, para esto realizaremos una función que, al dar la longitud del arreglo, nos dé un arreglo con números aleatorios, para poder hacerlo más práctico. Esta función la vimos en clases gracias a un compañero que nos la mostró.

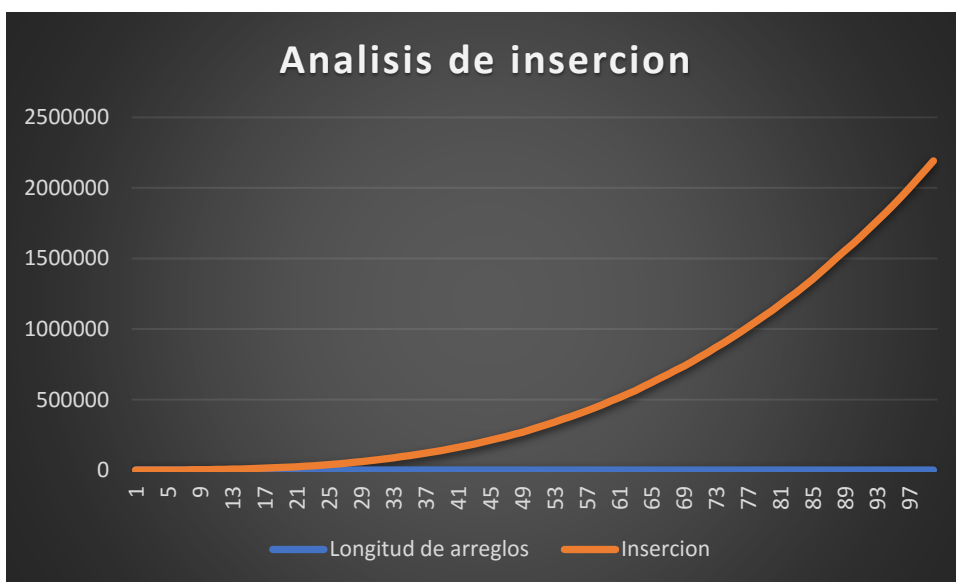
```
def arreglosgenerados(cantidaddearreglos):
    n=cantidaddearreglos
    for i in range(1, n+1):
        arri=random.sample(range(1,10000),10+5*(i-1))
        print(arri)
```

Con este algoritmo podemos crear arreglos aleatorios tales que me cuenten las operaciones que realizan cada uno de los algoritmos y así poder analizar los datos.

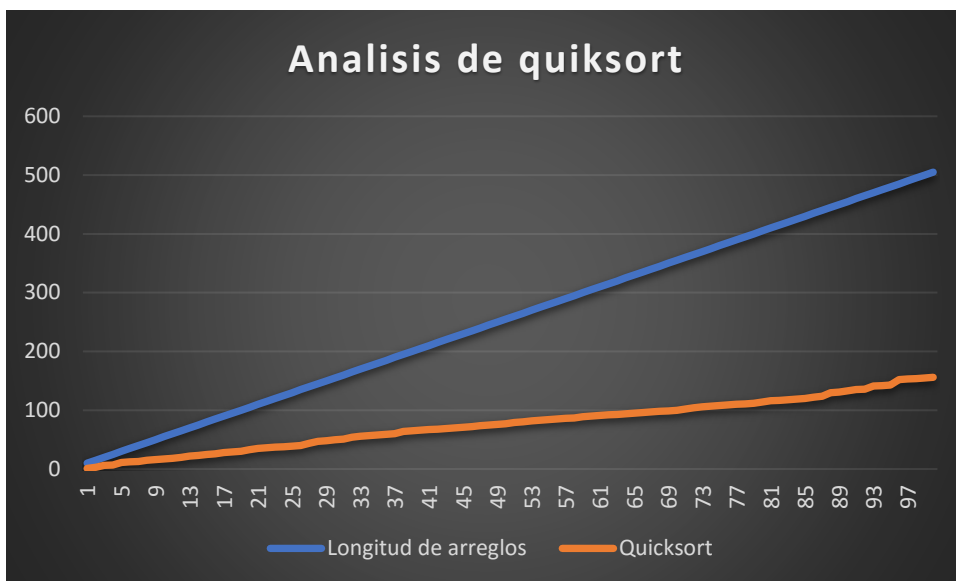
Para el caso de Selection tenemos que:



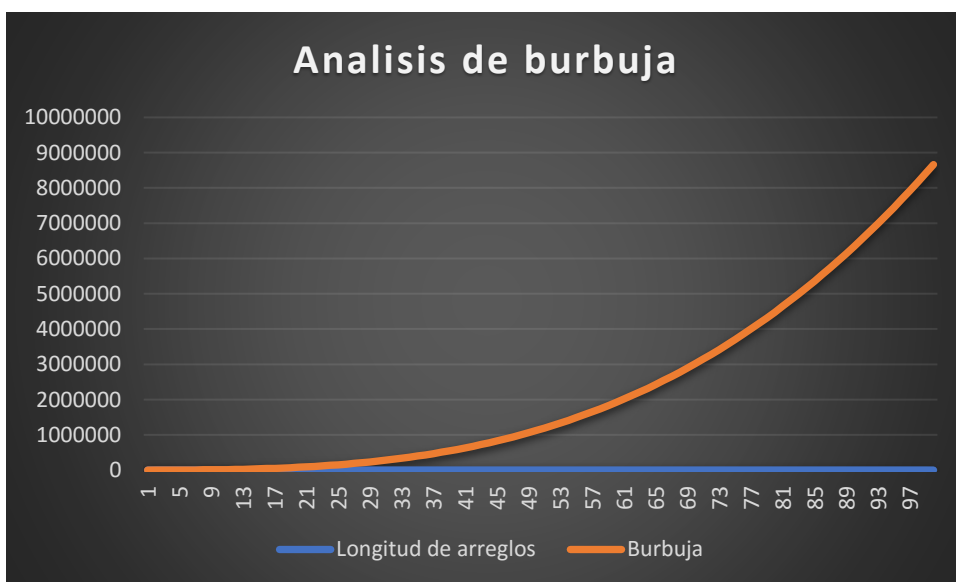
Para el caso de Insercion tenemos que:



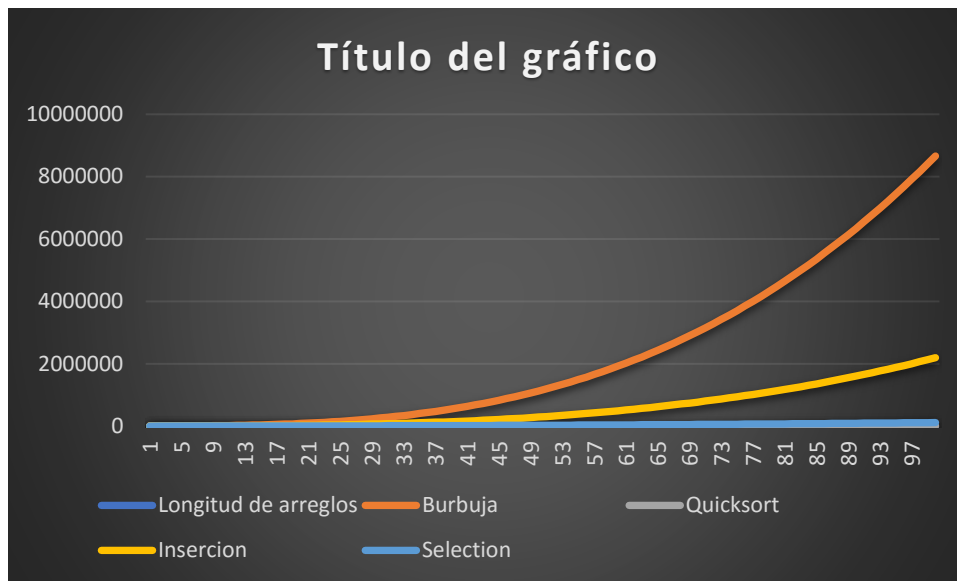
Para el caso de Quicksort tenemos que:



Para el caso de Quicksort tenemos que:



Por último, tenemos los algoritmos juntos en una tabla para analizar sus operaciones entre si.



Cada gráfica se hizo con el mismo algoritmo que mencioné al principio, los datos se juntaron y ahora están siendo comparados en esta última gráfica.

Como podemos observar, y ya lo habíamos mencionado, el algoritmo más eficiente es sin dudas el quicksort. Aunque los otros también pueden ser utilizados para ordenar los mismos arreglos,

Como conclusión, se puede notar que los algoritmos de ordenación son muy útiles para poder entender mejor lo que es la complejidad computacional. Así mismo, se puede decir que el mejor algoritmo de todos los cuatro es el de Quicksort, ya que muestra un mejor desempeño con el número de operaciones.

Por último, al graficar y tener el análisis de complejidad de cada uno, te ayuda a saber cuál es más eficiente que otro, esta herramienta también puede ser utilizada en otros tipos de algoritmos en un futuro.