Algoritmos de ordenación

Luis Enrique López Nerio Universidad Autonoma de Nuevo Leon
luiselopeznerio@gmail.com

17 de Septiembre del 2017

Abstract

En este reporte describiré brevemente algunos algoritmos de ordenación, la importancia de estudiar los algoritmos de ordenación reside en que son procesos que se presentan con gran frecuencia dentro de las ciencias computacionales ademas ayudan a introducir la idea de complejidad computacional. Dentro de la descripción se cubren aspectos como, la idea general del algoritmo, código del mismo, ventajas, desventajas en su uso, y finalmente su complejidad computacional.

I. Introduccíon

Un algoritmo es básicamente una seria de pasos claros y ordenados que nos ayudan a conseguir un resultado, dentro de un algoritmo se puede contar con datos de entrada que son tratados por el algoritmo y que producen una salida. El algoritmo debe ser los suficientemente claro y ordenado para producir el resultado esperado independientemente de los valores de entrada.

De manera más específica, un algoritmo de ordenación, es un proceso en el cual dado una serie de datos, se realizan pasos secuenciales que ordenan nuestros datos de una manera específica. Es importante notar que esta serie de datos debe tener la característica de poseer un orden, por ejemplo para los números reales podemos decir dados dos números a, b, si alguno de ellos es mayor a otro o si son iguales; sin embargo los datos que ordenaremos pueden no serán necesariamente números, podrían ser letras o cadenas de caracteres, por lo tanto es importante definir dados nuestros datos, que consideramos nosotros como orden.

Una vez definidos lo que es un algoritmo de ordenación ya podemos comenzar a estudiarlos más a fondo, para fines prácticos nuestros algoritmos de ordenación solo ordenaran números reales en orden ascendente, entonces nuestras información de entrada serán arreglos de números de cierta longitud y nuestros datos de salida serán el mismo arreglo ordenado en orden ascendente.

Si en nuestra vida diaria se nos presenta una serie de datos como en la figura 1, la forma en la que ordenaríamos seguramente sería algo muy intuitivo, observaríamos fácilmente que el valor mas pequeño es el 0 y que el más grande es el 23 y así sucesivamente ordenaríamos el arreglo, sin

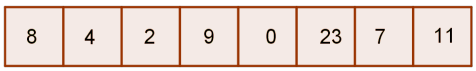

Figure 1: *ejemplo de un arreglo*

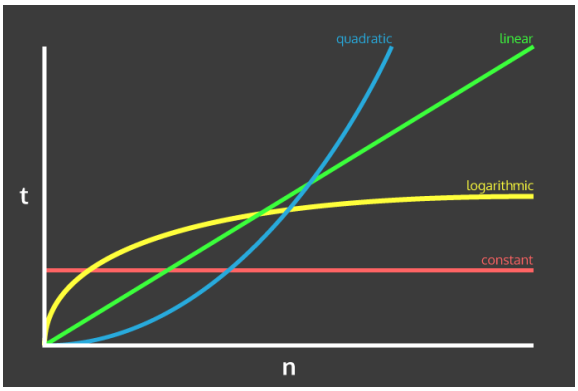

Figure 2: *Funciones de complejidad*

embargo esto no se traduce en un algoritmo, debemos tomar en cuenta que una computadora solo puede hacer una operación a la vez, además si quisiéramos ordenar un arreglo que contenga, por ejemplo, 1,000,000 de datos, este proceso no serviría con nuestra forma intuitiva de ordenar solo viendo e identificando los menores valores.

Esto nos hace observar que es necesario un proceso más ordenado para ordenar este arreglo, para nuestro reporte cubriremos 4 algoritmos de ordenación:

1) Burbuja
2) Inserción
3) Selección
4) Quicksort

II. Complejidad computacional

Para analizar nuestros algoritmos haremos uso de una herramienta para analizar su desempeño, alguna de las formas en que podemos analizar nuestros algoritmos es midiendo alguna variable como la cantidad de memoria utilizada, para nuestros fines utilizaremos como medida para medir el desempeño de nuestro algoritmo el tiempo que tarde en realizar la ordenación, más específicamente el número de operaciones primitivas que utiliza para realizar el ordenamiento dado el tamaño del valor de entrada.

Dado un algoritmo, utilizaremos la notación $\mathcal{O}$ para denotar el número de operaciones que toma nuestro algoritmo para realizar la ordenacion de los datos en el peor de los casos, dado el tamaño de los datos de entrada, por ejemplo si nuestro algoritmos de ordenación toma $n^2 + 4n + 3$ operaciones, diremos que tiene una complejidad $\mathcal{O}(n^2)$.

La notación $\mathcal{O}$, la podemos interpretar como una función la cual depende del la variable n, donde n es el tamaÃśo de la entrada, por ejemplo un algoritmo con complejidad $O(n^3)$, tendria una grafica que se comporta como una función polinómica cubica conforme sus datos de entrada crecen; algunos de los ejemplos de las complejidades computacionales mas comunes se muestran en la figura2.

III. Ordenamiento burbuja

El primer algoritmo que analizaremos será el ordenamiento burbuja, este ordenamiento es conocido por la simplicidad de su idea, intuitivamente es muy facil de entender, sin embargo es muy ineficiente para ordenar arreglos de longitud muy grande, a continuación veremos porque.

La idea general del algoritmo es: dado un arreglo de longitud n, repetidamente hacer cambios entre elementos adyacentes en nuestro arreglo, al final de recorrer el arreglo por primera vez nuestro valor máximo del arreglo se encontrara en su valor correspondiente, sin embargo el proceso de hacer cambios tendrá que realizarse hasta que todos los elementos estén en su orden por lo que es necesario realizar una iteración más desde el inicio del arreglo hasta el penúltimo elemento y así sucesivamente.

A continuación se presentaran el codigo del algoritmo burbuja, este algoritmo utiliza una función auxiliar para hacer un cambio de elementos en el arreglo:

```
def swap_b(arreglo, l,u):
    ##arreglo = arrelgo arbitrario,
    ##a,b= indices del arreglo a cambiar
    if arreglo[l]>arreglo[u]:
        aux=arreglo[l]
        arreglo[l]=arreglo[u]
        arreglo[u]=aux
        return 1
    else:
        return 0
```

Codigo 1.-Funcion para cambiar de lugar dos elementos de un arreglo

```
def burbuja(arreglo):
  ##Parametros
  ##arreglo = arreglo de longitud arbitraria
    swap_count = 1 ##Variable contadora de los cambios en arreglo
    operaciones=0 ##n\'umero de operaciones del algoritmo
    while True:
        swap_count=0
        for i in range(len(arreglo)-1):
            operaciones = operaciones + 1
            swap_count = swap_count + swap_b(arreglo, i,i+1)
        if swap_count == 0 :
            break
    return operaciones
```

Codigo 2.-Funcion para ordenar un arreglo con algoritmo burbuja

La complejidad del algoritmo bubble sort es de $\mathcal{O}(n^2)$ ya que en el peor de los casos, en el cual el arreglo está completamente invertido,se tendrá que recorrer el arreglo en el while (n-1) veces y tiene que realizar una iteración for dentro del while n veces. El algoritmo bubble sort por lo tanto no tiene un muy buen desempeño algunas ventajas son que, su interpretación es sencilla y no requiere espacio en memoria adicional, lo cual es de gran ayuda cuando el espacio en memoria es reducido.

IV. Ordenamiento por inserción

La idea del ordenamiento por inserción es tener dos arreglos, uno ordenado y otro desordenado; al comenzar consideramos el primer elemento ordenado, entonces iteramos sobre el arreglo desordenado tomando un elemento a la vez, cada elemento que tomemos lo insertaremos sobre el arreglo ordenado. Como no sabemos cuál debe ser el lugar del elemento desordenado en el arreglo ordenado entonces debemos iterar sobre el arreglo ordenado para insertar el elemento desordenado en su posición correspondiente; este proceso se realizara hasta que nuestros arreglo desordenado este vacío.

```
def insertion(arreglo):
    ##arreglo = arreglo de longitud arbitraria
    operaciones = 0##variable que cuenta operaciones del algoritmo
    for i in range(1, len(arreglo)):
        j = i
        operaciones = operaciones + 1
        while j > 0 and arreglo[j]< arreglo[j-1]:
            operaciones = operaciones + 1
            aux = arreglo[j]
            arreglo[j] = arreglo[j-1]
            arreglo[j-1]=aux
            j=j-1
    return operaciones
```

Codigo 3.-Funcion para ordenar un arreglo con algoritmo insertion

Hay varios aspectos a tomar en cuenta en el ordenamiento de inserción, sobre el mismo arreglo podemos realizar el proceso de ordenamiento, por lo tanto no requiere memoria adicional para su ordenación, al consistir de una lista ordenada y otra desordenada entonces puede ordenar listas conforme agregas elementos al arreglo original, esta situación se presenta con frecuencia por lo que el algoritmo insertion puede ser util en estos casos.

Como el ordenamiento de inserción tiene que iterar sobre todo el arreglo para ir tomando los elementos desordenados y a su vez iterar sobre un arreglo ordenado para insertar el elemento, entonces la complejidad computacional del algoritmo es de $\mathcal{O}(n^2)$

V. Ordenamiento por selección

La idea del algoritmo de selección consiste en tomar el primer elemento y seleccionarlo como el valor mínimo, una vez hecho esto se recorre el arreglo comparando el elemento en turno con el valor mínimo, si el elemento iterado es menor que el mínimo, se selecciona como el valor mínimo y se realiza esto hasta que se itera todo el arreglo, posteriormente se posiciona el valor mínimo en la primer posición, esto se repite para los valores restantes.

El algoritmo de selección al igual que el algoritmo de inserción ordena el arreglo en cada iteración que realiza, tampoco utiliza memoria adicional al ir almacenando los valores sobre el mismo arreglo existente.

```
def selection(arreglo):
    global operaciones##numero de operacines del algoritmo
    operaciones = 0
    for i in range(0, len(arreglo)-1):
```

```
        valor_minimo = i
        for j in range(i+1, len(arreglo)):
            operaciones = operaciones + 1
            if arreglo[j]< arreglo[valor_minimo]:
                valor_minimo = j
        if valor_minimo != i:
            aux = arreglo[i]
            arreglo[i] = arreglo[valor_minimo]
            arreglo[valor_minimo] = aux
            operaciones = operaciones + 3
    return operaciones
```

Codigo 4.-Funcion para ordenar un arreglo con algoritmo selection

Analizar la complejidad del algoritmo de selección no es muy complicado, al iterar sobre la longitud del arreglo y repetir este proceso todas las veces del número restante de elementos del arreglo, sabemos que su complejidad es $\mathcal{O}(n^2)$, al ordenar los elementos dentro del mismo arreglo lo hace muy conveniente para situaciones en las que no hay mucha memoria, sin embargo su desempeño es mediocre comparado con otros algoritmos.

VI. Ordenamiento Quicksort

El siguiente algoritmo que analizaremos será el algoritmo Quicksort, este algoritmo fue creado por Tony Hoare en el año 1959, este algoritmo entra de los que son clasificados de "divide y conquista", al poder dividir en problemas más pequeños del mismo tipo, por lo tanto este algoritmo usa recursión para ordenar el arreglo.

La idea básica del algoritmo consiste en elegir un elemento del arreglo, a este elemento elegido lo llamaremos elemento pivote, lo siguiente será reordenar el arreglo de manera que todos los elementos más pequeños del pivote estén del lado izquierdo de este y los mismo para los elementos mayores.

Una vez que termina de ordenar el arreglo en torno al pivote, el pivote se encontrara en su posición correspondiente, entonces se realizar el mismo proceso para la parte derecha e izquierda del arreglo que están sin ordenar.

Para fines prácticos al escribir el pseudocodigo utilizaremos la misma función de swap que se utilizó para el algoritmo de ordenación burbuja, y se declararan dos funcione, una llamada quicksort y otra llamada partición, que en conjunto realizan el ordenamiento.

```
def quicksort(arreglo,low,high ):
    ##Paremetros
    ##arreglo = arreglo longitud arbitraria
    ##low indice menor de arreglo
    ##indice mayor de arreglo
    global operaciones_q
    if low < high:
        m = particion(arreglo, low, high)
        quicksort(arreglo,low,m-1)
        quicksort(arreglo, m+1 , high)
    return operaciones_q

def particion(arreglo, low, high):
    ##Paremetros
```

```
    ##arreglo = arreglo longitud arbitraria
    ##low indice menor de arreglo
    ##indice mayor de arreglo
    global operaciones_q
    pivote = arreglo[high]
    wall = low-1
    for j in range(low, high):
        operaciones_q = operaciones_q + 5
        if arreglo[j]<pivote:
            wall = wall + 1
            swap(arreglo,wall,j)
    if(arreglo[high]< arreglo[wall+1]):
        swap(arreglo,wall+1,high)
    return wall+1
```

Codigo 5.-Funciones para ordenar un arreglo con algoritmo quicksort

Matemáticamente se puede demostrar que la complejidad del algoritmo quicksort es de $\mathcal{O}(nlog(n))$, aunque para los peores casos puede llegar a tomar $\mathcal{O}(n^2)$, hay esquemas en los que el valor pivote se elige tomando diferentes valores al azar y de esos valores tomar la mediana. Para arreglos que están hasta cierto punto ordenados o que contienen demasiados valores repetidos su desempeño disminuye.

VII. Experimentos

Con los codigos de los algoritmos de ordenación listos y con su complejidad computacional obtenidad de manera teorica el siguiente paso es provarlos en la practica y medir su desempeño, para esto en cada uno de los codigos se agregaron instrucciones que nos ayudan a medir aproximadamente el número de operaciones realizads. Ademas se escribio una función que nos permitira crear arreglos con números aleatorios y de una longitud deaseada, el codigo se presenta a continuación:

```
def ran_num(n,lim_inf=0, lim_sup=1000):
    arreglo = []
    for i in range(n):
        arreglo.append(random.randint(lim_inf, lim_sup))
    return arreglo
```

Codigo 6.- Funcion que crea arreglo aleatorio de longitud deseada

Con estos codigos ahora podemos hacer una prueba, crearemos arreglos de una longitud inicial, en este caso la longitud inicial sera 2, posteriormente copiaremos este arreglo 4 veces para que cada uno de los algoritmos ordene un arreglo con los mismos elementos, se imprimiran en pantalla la longitud del arreglo y el número de operaciones que le tomo a cada uno de los algoritmos ordenarlo, se repetira este proceso 30 veces, por lo tanto se ordenaran 30 arreglos de longitud 2 con los 4 diferentes algoritmos. Para finaliazar se repetira este proceso incrementando la longitud del arreglo en 500 hasta llegar a 10,000.

El codigo para realizar este proceso se presenta a continuación:

```

longitud = 2
print("Longitud arreglo", "Burbuja","Seleccion","Insertion", "Quicksort")
while (longitud < 10100):
    for repeticion in range(30):
        arreglo = ran_num(longitud)
        arreglo1, arreglo2, arreglo3, arreglo4 = copy.deepcopy(arreglo), copy.deepcopy(
    arreglo),copy.deepcopy(arreglo),copy.deepcopy(arreglo)
```

```
        bubble_op = burbuja(arreglo1)
        selection_op = selection(arreglo2)
        insertion_op = insertion(arreglo3)
        operaciones_q =0
        quicksort_op = quicksort(arreglo4,0, len(arreglo4)-1)
        print(longitud,bubble_op, selection_op,insertion_op,quicksort_op)

    longitud += 500
```

Codigo 6.- Codigo para crear base de datos con resultados de los algoritmos de ordenacion

Para finalizar la base de datos fue tratada con el lenguaje de programación R, se realizaron una grafica scatterplot la cual nos muestra en número de operaciones contra la longitud de arreglo .

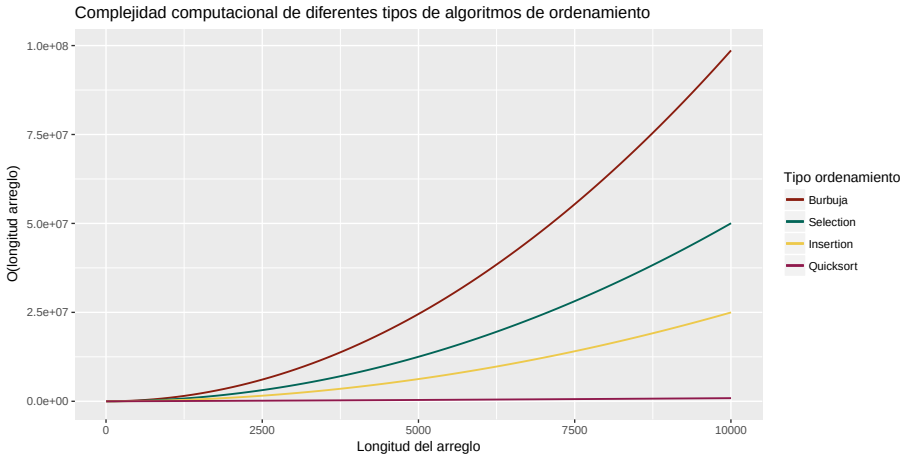

Figure 3: *Complejidad de algoritmos*

Las siguientes graficas nos muestran los diagramas de cajas para cada uno de los algoritmos de ordenación, los diagramas de cajas nos ayudan a observar a variaciÃşn que tiene el algoritmo para diferentes tamaños de arreglos, usualmente se espera mas variación cuando los arreglos son mas grandes.

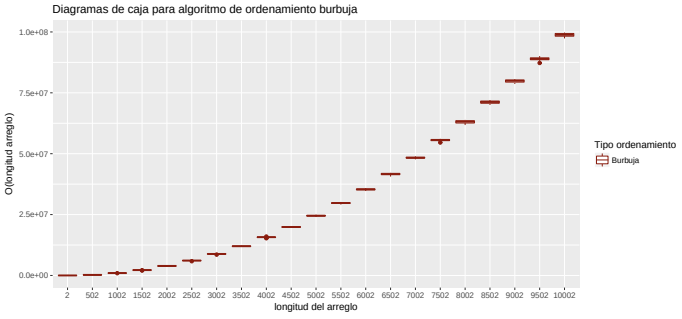

Figure 4: *Diagrama de caja Bubble*

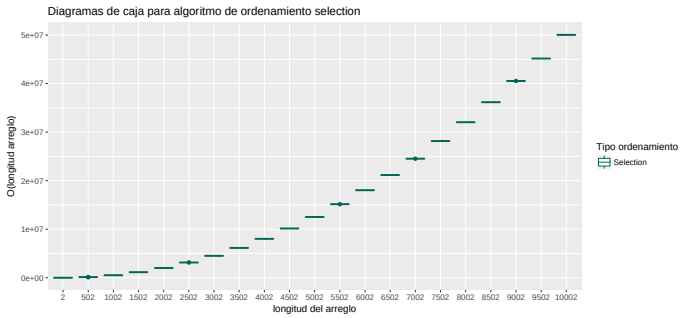

Figure 5: *Diagrama de caja Selection*

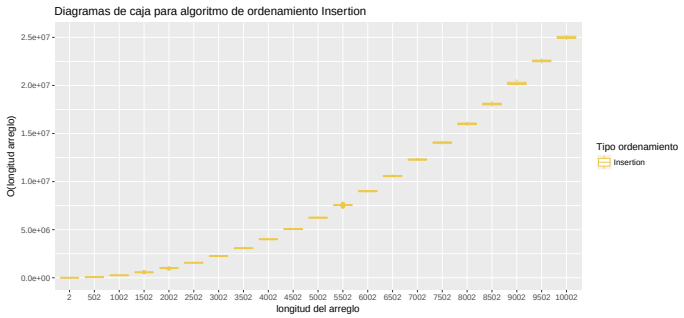

Figure 6: *Diagrama de caja insertion*

VIII. Conclusión

Mi conclusión que los algoritmos de ordenación son una gran herramienta para comprender mejor las herramientos de analis de algoritmos que son la complejidad computacional. Tales herramientas nos ayudaron a diferenciar algoritmos que cumplen con el mismo proposito.

Como conclusiÃşn sobre cual algoritmo de ordenaciÃşn es mejor la respuesta teorica nos decia que el algoritmo con mejor desempeÃśo es el algoritmo quiksort, esto se demostro de manera practica con el experimento realizado, ya que la grafica muestra una gran superioridad en cuanto a número de operaciones contra los otros algoritmos, el desempeÃśo mostrado los pone en el siguinete orden deacuerdo a su desemepeÃśo:

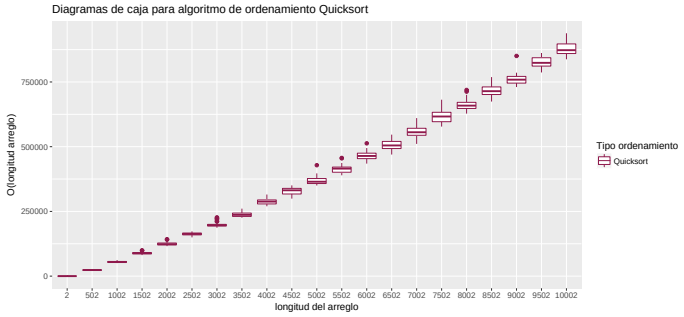

Figure 7: *Diagrama de caja Quicksort*

1 Quicksort
2 Insertion
3 Selection
4 Bubble

Ademas el diagrama de caja nos indica que los cuatro algoritmos no presentan gran variación en su desempeÃśo al ordenar arreglos de la misma longitud.Sin embargo la respuesta en cuanto al mejor algoritmo dependera de la situación ya que mi experimento no tomo en cuenta el espacio en memoria utilizado por cada algoritmo, porque para nuestro caso no era un impedimento, pero se pueden presentar casos en los que la memoria sea limitada y el uso del algoritmo quicsort no sea tan recomendable contra los otros 3.

También se debe tomar en cuenta que el diseÃśo del experimento utilizo arreglos aleatorios y no arreglos con un cierto grado de orden que pueden ocasionar que el algoritmo quicksort no tenga muy buen desempelo contra los otros tres, como el insertion o selection, que suelen ser buenas alternativas para arreglos con partes ya ordenadas. Asi que como trabajo futuro queda diseÃśar experimentos que resalten las ventajas de los diferentes algoritmos y en cuanto al desempeÃśo del número de operaciones podemos concluir que el algoritmo Quicksort es la mejor alternativa de las cuatro. El algoritmo bubble sort casi siempre es la manera incorrecta de atacar la ordenación de un arreglo muy grande, como alguna vez lo dijo el mejor presidente de la historía de los Estados Unidos de America[1].

[1]Entrevista a Obama en Google