Algoritmos de ordenación

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

1 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 y ayudan a introducir la idea de complejidad computacional. Dentro de la descripción se cubren aspectos como, la idea general del algoritmo, pseudocódigo del mismo, ventajas y 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 ordenados 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 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

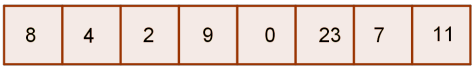

Figure 1: *ejemplo de un arreglo*

nuestras información de entrada serán arreglos de números de cierta longitud y nuestros datos de salida será 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 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:

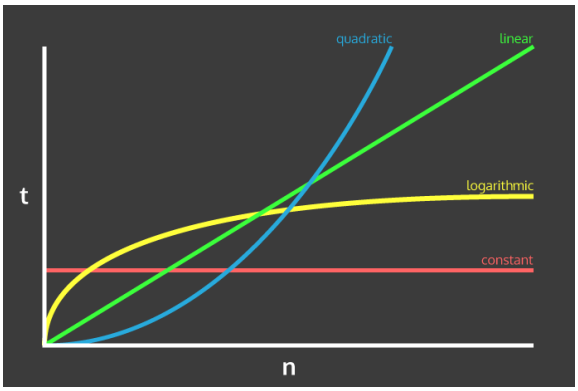

Figure 2: *Funciones de complejidad*

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 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 tamaÃśo de los datos de entrada, por ejemplo si nuestro algoritmos de ordenación toma $n^2 + 4n + 3$ operaciones, diremos que tiene una complejidad $O(n^2)$.

La notación O mayuscula 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 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.

Algorithm 1 $swap(arr, l, u)$:

```
if arr[l] > arr[u] then
    aux = arr[l]
    arr[l] = arr[u]
    arreglo[u] = aux
    return 1
else
    return 0
end if
```

La complejidad del algoritmo bubble sort es de $O(n^2)$ ya que en el peor de los casos en el cual el arreglo está completamente invertido, en este caso 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

Algorithm 2 *Bubble*(arr) :

```
swapcount = 1
while TRUE do
  swapcount = 0
  for i = 0 to londitud(arr) − 1 do
    swapcount = swapcount + swap(arr, i, i + 1)
  end for
  if swapcount == 0 then
    break
  end if
end while
```

muy buen desempeño algunas ventajas son que, su interpretación es sencilla y no requiere espacio en memoria adicional.

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.

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 las recibe con otras previamente ordenadas.

Como el ordenamiento de inserción tiene que iterar sobre todo el arreglo para ir tomando los elementos desordenadas y a su

Algorithm 3 *insertion*(arr)

```
for i = 1 to length(arr) − 1 do
  j = i
  while j > 0 AND arr[j] > arr[j − 1]
  do
    aux = arr[j]
    arr[j] = arr[j − 1]
    arr[j − 1] = aux
    j = j − 1
  end while
end for
```

vez iterar sobre un arreglo ordenado para insertar el elemento, entonces la complejidad computacional del algoritmo es de $O(n^2)$

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.

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 $O(n^2)$, 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.

Algorithm 4 *selection*(arr)

```
for i = 1 to length(arr) − 1 do
  val_minimo = i
  for j = i + 1 to len(arr) do

    if arr[j] < arr[val_minimo] then
      val_min = j
    end if
  end for
  if i ≠ val_minimo then
    aux = arr[val_minimo]
    arr[val_minimo] = arr[i]
    arr[i] = aux
  end if
end for
```

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 âĂIJdivide y conquistaâĂİ al poder dividir en problemas más pequeños del mismo tipo, por lo tanto este algoritmo usa recursión para funcionar.

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.

Algorithm 5 *quicksort*$(arr, low, high)$

```
if low < high then
  m = particion(arr, low, high)
  quicksort(arr, low, m − 1)
  quicksort(arr, m + 1, high)
end if
```

Algorithm 6 *particion*$(arr, low, high)$

```
pivote = arr[high]
Wall = low − 1
for j = low to high − 1 do
  if arr[j] < pivote then
    wall = wall + 1
    swap(arr, wall, j)
  end if
end for
if arr[high] < arr[wall + 1] then
  swap(arr, high, wall + 1)
end if
return wall + 1
```

Matemáticamente se puede demostrar que la complejidad del algoritmo quicksort es de $O(nlog(n))$, aunque para los peores casos puede llegar a tomar $O(n^2)$, 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. Conclusión

Para finalizar me gustaria concluir 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 mi respuesta seria la siguiente:

Si lo que quieres es velocidad el mejor algoritmo de los cuatro revisados es el algoritmo Quicsort ya que tiene una complejidad

computaciÃşnal de $O(nlog(n))$, incluso como futuro modificaciÃşn se puede desarrollar la elección de un mejor pivote para mejorar su rendimiento por lo que hay un area de oportunidad.

Si se cuenta con arreglos que se van ordenando conforme llega información, insertion sort sería el mas adecuado. Si se cuenta con poca memoria el algoritmo se selección sera la opción mas viable. 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