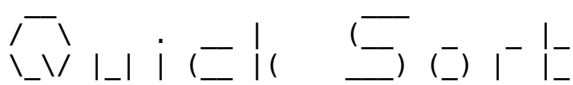
"Análisis de Algoritmos"
José Helo Guzmán



Contenido

▬▬▬▬ 1. Introducción.

Ocasionalmente se encuentra uno con problemas sumamente difíciles de resolver. O bien problemas que todavía no tienen una solución algorítmica. Cuando un buen algoritmista, o programador, se ve en esta situación, existen un conjunto de técnicas que se pueden utilizar.

En este capítulo veremos nuevamente la técnica de “divide and conquer”. Aplicaremos esta técnica a un problema de ordenamiento. Se resolverá el problema del “quick sort” por medio de un procedimiento muy elegante y muy utilizado en la práctica.

Por último es necesario mencionar que algunos algoritmos parecen naturalmente diseñados para programarse por medio de iteración, otros, parecen que son más fáciles de implementar por medio de recursión.

▬▬▬▬ 2. Divide and Conquer.

Esta técnica para resolver problemas se puede aplicar en una gran variedad de situaciones. De ahora en adelante, cada vez que se enfrente a un problema se puede hacer la pregunta si es resoluble mediante este método.

Para presentarla, trataremos de resolver el problema de ordenar una lista de elementos, cosa que ya hemos hecho con anterioridad. Pero en esta ocasión se utilizará un algoritmo muy elegante y eficiente.

▬▬▬ 3. Quick Sort.

El “quick sort” es un algoritmo de ordenamiento. Por ejemplo, el lenguaje de programación “C” en su biblioteca estándar tiene una función llamada “qsort” que implementa este algoritmo. En este caso deseamos ordenar la lista del menor elemento al mayor elemento.

El algoritmo del “quick sort” es el siguiente.

1. Escoja un elemento de pivote.
   Después veremos algunas técnicas para escogerlo,
   por el momento escogeremos el primer elemento.

2. Haga una partición de la lista en dos sublistas.
   Una lista de elementos menores al pivote.
   Una lista de elementos mayores al pivote.

4. Recursivamente ordene la lista de elementos menores y de elementos mayores.

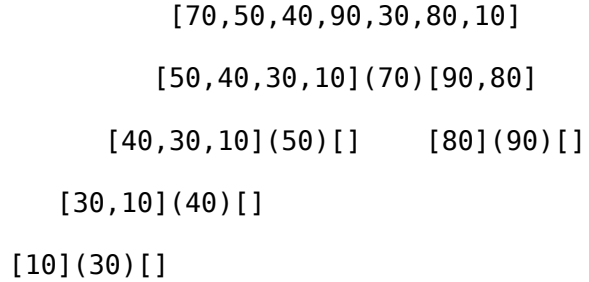
5. Ordene las listas de la siguiente manera:
   listaMenoresOrdenada + pivot + listaMayoresOrdenada.

●● Ejemplo del “quick sort”.

Se tiene la siguiente lista de elementos:

[70, 50, 40, 90, 30, 80, 10]

El quick sort realizará las siguientes llamadas:



Para la reconstrucción de la lista ordenada se debe seguir en cada nodo la forma:

```
listaMenores + pivot + listaMayores
```

▬▬▬▬ 4. Tiempo de Ejecución.

El desempeño del “quick sort” depende fuertemente de la forma como se escoja el pivote. Tal como hemos supuesto, si se escoge el primer elemento de la lista como pivote, puede ocurrir que se tenga que recorrer una lista ordenada.

A continuación hacemos un estudio detallado de su tiempo de ejecución.

▬▬▬▬ 5. El Peor Caso.

El peor caso se presenta cuando se tiene una lista ordenada de forma inversa a lo que se desea, de mayor a menor:

```
                          [8,7,6,5,4,3,2,1]
                        [7,6,5,4,3,2,1](8)[]
                      [6,5,4,3,2,1](7)[]
                    [5,4,3,2,1](6)[]
                 [4,3,2,1](5)[]
              [3,2,1](4)[]
          [2,1](3)[]
      [1](2)[]
[](1)[]
```

En este caso la lista se encuentra ordenada de menor a mayor:

```
[1,2,3,4,5,6,7,8]
[](1)[2,3,4,5,6,7,8]
     [](2)[3,4,5,6,7,8]
          [](3)[4,5,6,7,8]
               [](4)[5,6,7,8]
                    [](5)[6,7,8]
                         [](6)[7,8]
```

[](7)[8]

[](8)[]

Observemos que el número de niveles que se produce en este caso son 8. Tantos niveles como elementos tiene la lista. Por lo tanto tienen un tiempo de O(n).

Veamos ahora como se construyen las particiones:

```
[1,2,3,4,5,6,7,8]      -----> Para construir las sublistas
                       -----> se deben recorrer todos
[](1)[2,3,4,5,6,7,8]   -----> los elementos, tiempo O(n)

     [](2)[3,4,5,6,7,8]

          [](3)[4,5,6,7,8]

               [](4)[5,6,7,8]

                    [](5)[6,7,8]

                         [](6)[7,8]

                              [](7)[8]
```

Observemos ahora como se construyen las particiones. Para construir una partición de debe recorrer toda la lista. De esta forma, también tiene un tiempo de O(n).

Para reconstruir las listas ordenadas, también se requiere un tiempo de O(n).

Por lo tanto, en el peor caso este algoritmo tiene tiempo $O(n)*O(n) = O(n^2)$

▬▬▬ 6. El Mejor Caso.

El largo del proceso anterior se debe a una mala escogencia del punto de pivote. Si se tomara un punto de pivote que equilibra la lista de menores, con la lista de mayores se requieren muchos menos llamadas a la pila. Observe el proceso con otro pivote que mantiene las búsquedas equilibradas se realizan únicamente 4 niveles.

```
  [1,2,3,4,5,6,7,8]

  [1,2,3](4)[5,6,7,8]

[1](2)[3]   [5](6)[7,8]

                 [](7)[8]
```

En este caso se puede observar que el número de niveles está dado por $O(\log_2(n))$. Pues en cada nivel se divide la lista en dos partes casi iguales.

Ahora veamos el primer nivel. Se escoge un elemento adecuado como pivote y los demás elementos se dividen en sublistas. Para construir las sublistas, también se debe de recorrer todos los elementos. Se deben recorrer los todos los 8 elementos. Así que esta operación toma tiempo O(n).

```
  [1,2,3,4,5,6,7,8]      -----> Para construir las sublistas
                         -----> se deben recorrer todos
  [1,2,3](4)[5,6,7,8]    -----> los elementos, tiempo O(n)

[1](2)[3]   [5](6)[7,8]

                 [](7)[8]
```

Por lo tanto el tiempo del mejor caso está dado por:
$O(n)*O(\log_2(n)) = O(n*\log_2(n))$

▬▬▬ 7. El Caso Promedio.

El “quick sort” es un caso muy interesante porque se han estudiado los tres casos posibles, el peor caso, el mejor caso y el caso promedio.

El caso promedio se acerca mucho al mejor caso si se escoge de manera adecuada el punto de pivote. Sin embargo, esta escogencia no puede utilizar mucho tiempo computacional o elevará el tiempo de ejecución final. Por esta razón se han propuesto algunas maneras para encontrar el pivot.

- Se puede escoger de forma aleatoria.
  Esto ha resultado un buena forma de acercarse el mejor caso.

- Se sacan el primer elemento, el elemento central y el último elemento.
  Se ordenan los tres elementos de menor a mayor.
  Se toma como pivote el valor que está en el centro.

▬▬▬ 8. Análisis del Tiempo de Ejecución.

Si utilizamos un proceso recursivo, se puede calcular el tiempo de ejecución mediante la siguiente relación de recurrencia:

$T(n) = T(k) + T(n-k-1) + n$

Los elementos T(k) y T(n-k-1), son para las llamadas recursivas y el último elemento es el tiempo que se tarde haciendo la partición.

El valor de “k” indica el número de elementos menores que el pivot.

●● El Peor Caso.

En el pero caso, tenemos que la fórmula se convierte en:

$T(0) = 1$

$T(n) = T(0) + T(n-0-1) + n$

$T(n) = T(0) + T(n-1) + n$

$T(n) = 1 + T(n-1) + n$

Entonces podemos plantear la relación como:

$T(0) = 1$

$T(n) = T(n-1) + n + 1$

Al resolver la relación de recurrencia se obtiene:

$$T(n) = \left(\frac{(n+1)*(n+2)}{2} \right)$$

Por lo que:

$T(n) = O(n^2)$

●● El Mejor Caso.

Si el punto de pivot se escoge correctamente, se puede obtener el mejor caso, donde el tiempo T(n) estará dado por:

$T(0) = 1$

$T(n) = T(n/2) + T(n/2) + n$

$T(n) = 2*T(n/2) + n$

Al resolver esta relación de recurrencia se obtiene:

$T(n) \sim O(n*\log_2(n))$ conforme $n \to +\infty$

▬▬▬ 9. “Merge Sort” y “Quick Sort”.

Cuando se presentó el algoritmo de “merge sort”, se estableció que su tiempo es de:

$O(n * \log_2(n))$

Esto parece indicar que es una mejor opción que el “quick sort”. Sin embargo el “quick sort” es un caso interesante. Hemos estudiado que tiene los siguientes tiempos.

En el pero caso: $O(n^2)$

En el mejor caso: $O(n * \log_2(n))$

En este momento cabe preguntarse cómo se comporta en el caso promedio. Vamos a discutir este tema.

Si tenemos la siguiente lista:

[2, 4, 6, 8, 10]

Suponga que tiene un código que imprime cada número.

imprimir: 2 4 6 8 10

Claramente este código es O(n) pues pasa por toda la lista para imprimirla.

Supongamos ahora que tenemos un código que también imprime la lista, pero hace una pausa de 1 segundo después de imprimir cada número.

imprimirPausa: 2 <1seg> 4 <1seg> 6 <1seg> 8 <1seg> 10 <1seg>

Esta función también toma tiempo O(n).

Sin embargo, en la práctica, uno prefiere el primer programa de imprimir pues es mucho más rápido y no realiza pausas. Entonces, aunque ambos sean O(n) uno prefiere usar el segundo programa. Cuando uno indica que el programa tiene un tiempo O(n) la definición inicial nos indicaba que el tiempo de ejecución es menor que:

$O(n) = c*n$ para algún valor de “c”

Por esta constante podría ser de 0.001 segundos para la primera función de imprimir y de 1 segundo para la segunda función. Es decir:

0.01*n para el programa de imprimir

1*n para el programa de imprimirPausa

El valor de esta constante no es importante cuando los algoritmos pertenecen a diferentes grupos de tiempo la Gran O. Por ejemplo, si un algoritmo es lineal y el otro es logarítmico, la constante no tiene importancia. Supongamos que se tienen dos algoritmos A y B, con las siguientes constantes:

Algoritmo A: tiempo (0.01 segs) *n

Algoritmo B: tiempo (1 seg) * $\log_2(n)$

Al ver únicamente la constante, uno podría suponer que el algoritmo A es mucho más rápido que el B. Supongamos que se ejecuta una lista con 4 billones de elementos, es decir una lista con 4,000,000,000. Entonces cada algoritmo tardaría los siguiente:

Algoritmo A: 0.01 * 4 billones = 463 días.

Algoritmo B: 1 * $\log_2$(4 billones) = 1 * 32 = 32 segundos.

Es evidente, que a pesar de la diferencia entre el valor de las constantes, el algoritmo B es mucho más rápido. Por lo tanto, se puede ignorar la constante.

Pero, en algunos casos, cuando el valor de la BigOh es muy cercano, las constantes son importantes. El “quick sort” y el “merge sort” son un ejemplo. La constante del “quick sort” es mucho más pequeña que la del “merge sort”. Por lo tanto, si ambos se pueden programar en un tiempo de O(n*$\log_2$(n), el “quick sort” es más rápido. Además, en la práctica, el “quick sort” recibe entradas en el caso promedio, muchas más veces que en el peor caso.

Tal como se discutió con anterioridad, para que el “quick sort” se acerque al caso de O(n*$\log_2$(n)), se debe escoger de forma apropiada el punto pivot del algoritmo.

▬▬▬ 10. Resumen.

• El método de dividir y conquistar trata de reducir un problema en problemas más pequeños.

• Si se implementa el “quick sort” y se escoge como punto de pivote el primer elemento de la lista, su tiempo de ejecución será de O(n^2).

• Si se escoge el punto de pivote de forma aleatoria, el tiempo promedio de ejecución será de O(n*$\log_2$(n)).

• En algunos algoritmos es importante considerar la constante que se elimina en la notación de la BigOh.

▬▬▬ 11. Ejercicios.

●● Ejercicio 1.

Escriba un programa para sumar los elementos de una lista.
Calcule el valor de la gran O para este programa.

●● Ejercicio 2.

Escriba un programa recursivo para sumar los elementos de una lista.
Calcule el valor de la gran O para este programa.

●● Ejercicio 3.

Dada una lista de números de tamaño arbitrario de la forma:

lista: [10, 20, 50, 40, 30]

Construya un algoritmo recursivo de “divide and conquer” para encontrar el máximo elemento de la lista. Cuál es el valor de la gran O de este programa?

●● Ejercicio 4.

Recuerda el algoritmo de “binary search” que vimos anteriormente?

Este es un algoritmo de “divide and conquer”, puede programar este algoritmo de forma recursiva?

●● Ejercicio 5.

Cuál es el tiempo de ejecución en notación de O grande para imprimir cada uno de los valores de una lista?

●● Ejercicio 6.

Cuál es el tiempo de ejecución en notación de O grande para elevar al cuadrado cada uno de los elementos de una lista?

●● Ejercicio 7.

Cuál es el tiempo de ejecución en notación de O grande para elevar al cuadrado el primer elemento de una lista?

●● Ejercicio 8.

Cuál es el tiempo de ejecución en notación de O grande para multiplicar todos los elementos de una lista.

Suponga que tiene la lista
[2,3,7,8,10].

Primero debe multiplicar cada valor de la lista por 2:
[4,6,14,16,20]

Después debe multiplicar cada valor de la lista por 3:
[6,9,21,24,30].

Después debe multiplicar cada valor de la lista por 7:
[14,21,49,56,70]

Y así sucesivamente hasta que se acaben los elementos de la lista.