Estructuras de datos y Algoritmos II

UNIDAD 1: Algoritmos de ordenamiento

Profesor: Ing. Jonathan Roberto Torres Castillo

Ingenieria en Computación

Universidad Nacional Autónoma de México

Tercera sesión

1. Recursividad
2. Introducción QuickSort.
3. Descripción del Algoritmo Quick Sort.
4. Ejemplos y aplicaciones.

Recursividad

Es definir algo en términos de si mismo.

Cualquier caso de definición recursiva o invocación de un algoritmo recursivo tiene que reducirse a la larga a alguna manipulación de uno o varios casos mas simples no recursivos.

Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas a si mismo. Claro que cualquier algoritmo que genere tal secuencia no termina nunca. Una función recursiva f debe definirse en términos que no impliquen a f al menos en un argumento o grupo de argumentos. Debe existir una "salida" de la secuencia de llamadas recursivas, sin esta salida no puede calculase ninguna función recursiva.

Algoritmos de ordenamiento: QuickSort

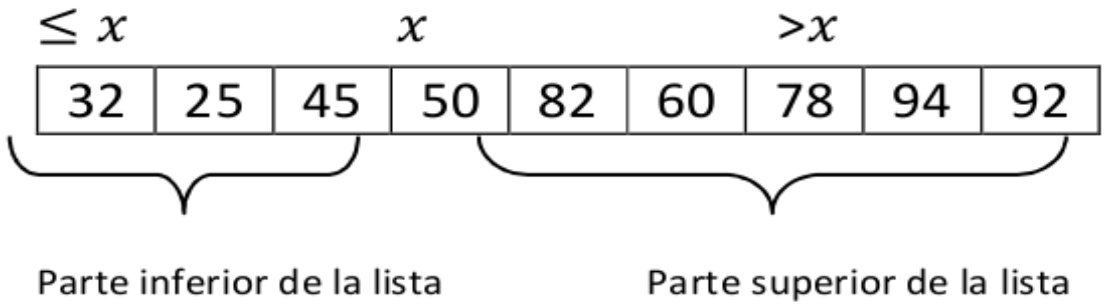
Algoritmo de ordenación 'QuickSort': Descripción y características

El algoritmo de ordenamiento rapido (*Quick Sort* en inglés) es un algoritmo de ordenamiento que tambien esta basado en la técnica divide y vencerás, es uno de los algoritmos mas utilizados en la computación. Fue inventado en *1960* por **Tony Hoare** cientifico britanico en computacion.

QuickSort: Dividir

El algoritmo de Quick Sort parte del mismo principio de **'Divide y vencerás'** solo que en este caso se tiene una forma de ordenamiento diferente y mucho mas rápida (recursividad).

1. Se divide un arreglo en 2 sub-arreglos utilizando un elemento **pivote** de manera que de un lado queden todos los elementos menores o iguales a él y del otro los mayores.

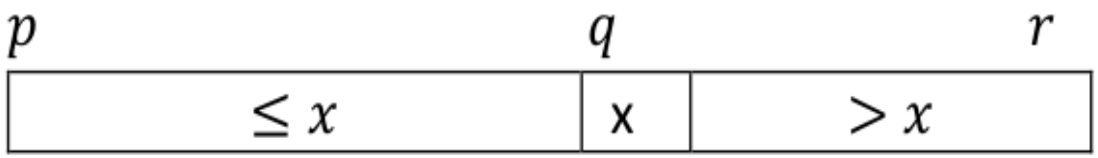


QuickSort: Dividir

Si el arreglo se representa como $A[p, ..., r]$ donde el primer elemento está en la posición p y el ultimo en la posición r, al dividir la lista se tiene que:

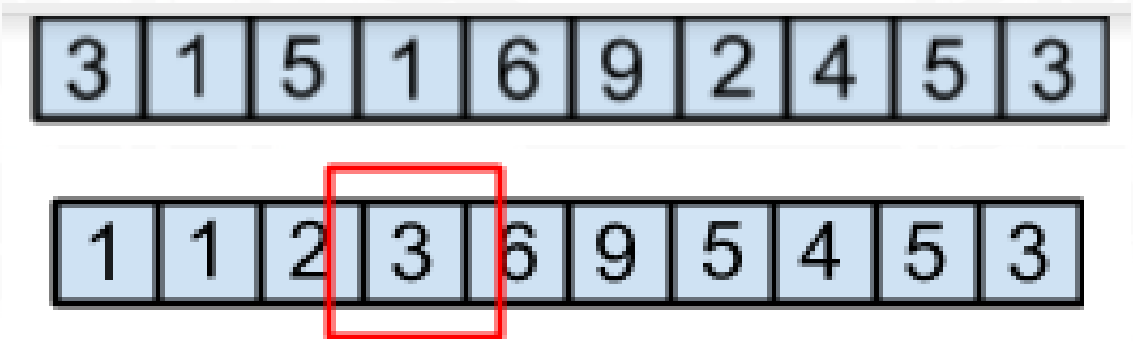
1. El elemento pivote es $x = A[q]$
2. La lista de la izquierda es $A[p, ..., q-1]$
3. La lista de la izquierda es $A[q+1, r]$

Los valores de x cumplen que $A[p, ..., q-1] \leq x < A[q+1, ..., r]$



QuickSort: Dividir y ordenar (recursividad)

El siguiente paso se refiere a agrupar a todos aquellos números menores al pivote en las primeras $q-1$ posiciones y los números mayores o iguales al pivote se agrupan a partir de las posición $q+1$. La siguiente figura ilustra este caso:



En este momento, tenemos que los números menores a 3 están ubicados antes de la cuarta posición y los mayores o iguales a 3 a partir de la quinta posición.

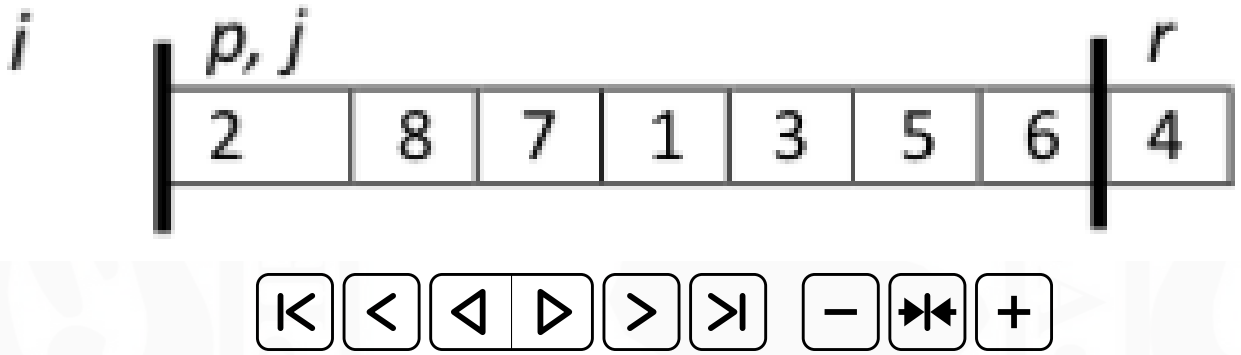
QuickSort: Dividir y ordenar (recursividad)

A continuación se muestra el paso recursivo, en donde el algoritmo se aplica tanto para el sub-arreglo definido por los índices 1 y 3, como para el sub-arreglo definido por los índices 5 y 10. Si hacemos las llamadas recursivas sobre los sub-arreglos subsecuentes hasta que su tamaño sea 1, tenemos el siguiente resultado:

QuickSort: Resultado

En el caso de un arreglo, como las sub-listas son parte del arreglo A y cada una ya está ordenada no es necesario combinarlas, el arreglo $A[p, ..., r]$ ya está ordenado.

Y así, finalmente, obtenemos el siguiente arreglo ordenado.


i
p, j
r
2
8
7
1
3
5
6
4

QuickSort

Un algoritmo general se puede representar como:

```
QuickSort()
Inicio
Si lista tiene mas de un elemento
 Particionar la lista en dos sublistas
 %(Sublista Izquierda y Sublista Derecha)
 Aplicar el algoritmo QuickSort() a Sublist Izquierda
 Aplicar Algoritmo QuickSort() a Sublista Derecha
 Combinar las 2 listas ordenadas
Fin Si
FIN
```

QuickSort

El algoritmo esta dividido básicamente en dos partes:

```
QuickSort(A,p,r)
Inicio
    Si p < r entonces
    // Si la lista tiene mas de un elemento
    q =Particionar(A,p,r)
    QuickSort(A,p,q-1)
    QuickSort(A,q+1,r)
Fin Si
Fin
```

QuickSort

```
Particionar(A,p,r)
Inicio
x=A[r]
i=p-1
para j=p hasta r-1
    Si A[j]<=x
        i=i+1
        intercambiar A[i] con A[j]
    Fin Si
Fin para
intercambiar A[i+1] con A[r]
retornar i+1
Fin
```

```
#Autor | Elba Karen Sáenz García
def intercambia( A, x, y ):
    tmp = A[x]
    A[x] = A[y]
    A[y] = tmp

def Particionar(A,p,r):
    x=A[r]
    i=p-1
    for j in range(p,r):
        if (A[j]<=x):
            i=i+1
            intercambia(A,i,j)
    intercambia(A,i+1,r)
    return i+1


def QuickSort(A,p,r):
    if ( p < r ):
        q=Particionar(A,p,r)
        print(A[p:r])
        QuickSort(A,p,q-1)
        QuickSort(A,q+1,r)
```

Complejidad del Algoritmo QuickSort

Eficiencia – Arreglo desordenado

Archivo de tamaño n es una potencia de 2, digamos $n = 2^m$, por lo que $m = log_2 n$. Asumiendo también que la posición correcta para el elemento pivote siempre cae en la mitad del subarreglo. En ese caso habrá aproximadamente n comparaciones (en realidad $n-1$) en la primera pasada, después de eso el archivo se parte en dos subarchivos de tamaño $n/2$, aproximadamente. Para cada uno de estos dos archivos se hacen aproximadamente $n/2$ comparaciones, y un total de 4 archivos de tamaño $n/4$. Después de dividir los subarchivos m veces, habrá n archivos de tamaño 1. Así que el número total de comparaciones para la ordenación completa es aproximadamente:

$$n + 2(n/2) + 4(n/4) + 8(n/8) + ... + n(n/n)$$

o

$$(n + n + n + ... + n)m$$

Complejidad del Algoritmo QuickSort

Eficiencia – Arreglo desordenado

Son m términos por que el archivo se subdivide m veces. Así que el número total de comparaciones es $O((n)(m))$ o O(n log(n)) (recordando que $m = log2n$).
(Tomamos el primer elemento del arreglo como pivote y este encontrara su posicion en la mitad del subarreglo)

Complejidad del Algoritmo QuickSort

Eficiencia – Arreglo ordenado

Tomamos el primero elemento del sub-arreglo (que ya está en su posición correcta), entonces el archivo original se divide en dos sub-archivos de tamaño 0 y $n-1$. Si el proceso continúa, se obtendrán un total de $n-1$ sub-archivos. El primero de tamaño n, el que sigue de $n-1$, el siguiente de tamaño $n-2$, y así sucesivamente. Asumiendo k comparaciones para reordenar un archivo de tamaño k, el número total de comparaciones para ordenar el archivo completo es:

$$n + (n–1) + (n–2) + ... + 2$$

que es $O(n^2)$. De manera similar si el archivo está ordenado en orden descendente se hacen particiones muy des-balanceadas. Así que la versión original de QuickSort tiene la absurda propiedad que **trabaja mejor** con archivos que están **completamente desordenados** y peor para archivos que están **completamente ordenados**. Opuesto a "la burbuja".

Referencias I

Thomas H Cormen.

Introduction to algorithms.

MIT press, 2009.

Donald Knuth.

The art of programming.

ITNow, 53(4), 2011.

Elba Karen Sáenz García.

Guía práctica de estudio 1, algoritmos de ordenamiento parte 1.

Facultad de ingeniería, UNAM.