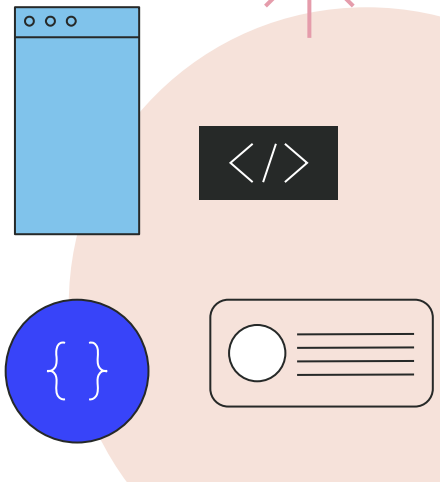
Estructura de datos y algoritmos

Ordenamiento y búsqueda Parte 2

Viviana Gasull



Contenidos

Metodos de ordenamiento

- **Bucket Sort**
- **Quick Sort**
- **Búsqueda**
- **Ordenamiento de objetos**

Bucket Sort

Es probablemente la distribución más simple que tiene un algoritmo. El único requerimiento esencial es que el tamaño del universo del cual los elementos estén siendo ordenados, sea constante.

Por ejemplo suponga que estamos ordenando elementos en el rango de {0, 1, . . ., m-1}. El ordenamiento de Bucket sort utiliza m contadores el contador i ésimo mantiene cuenta del número de ocurrencias del elemento i ésimo en el universo.

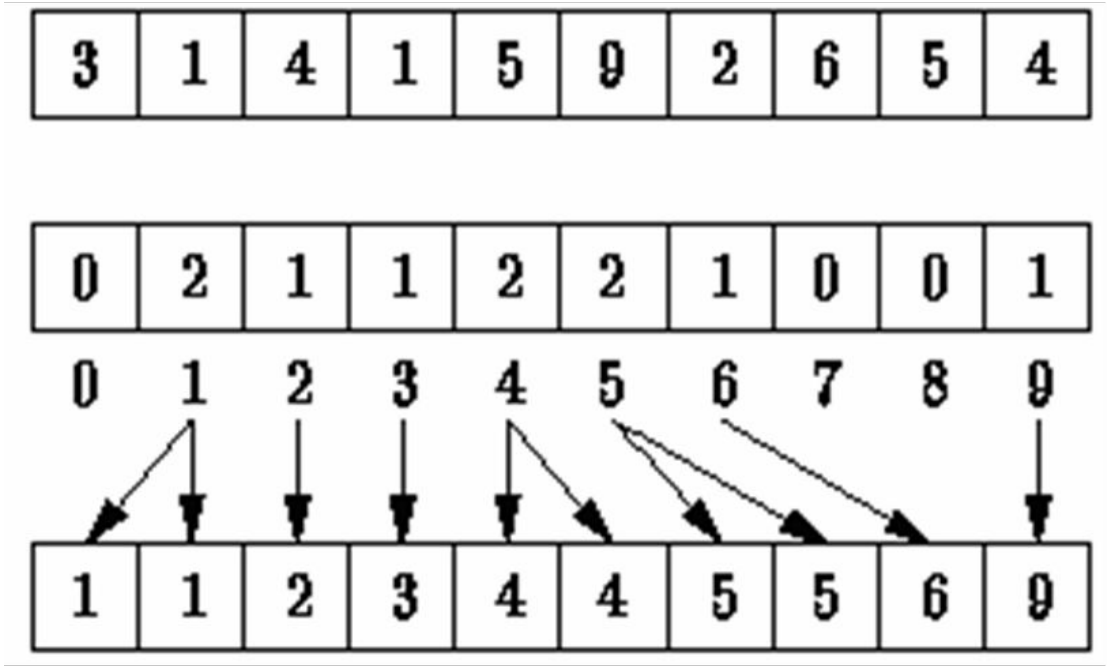
El algoritmo trabaja partiendo un arreglo en un número finito de buckets y luego ordena cada bucket.

Bucket Sort

Pasos del algoritmo:

1. Establece un arreglo vacío de buckets, del tamaño del rango a ordenar
2. Recorre el arreglo original incrementando en 1 el bucket correspondiente al elemento encontrado
3. Coloca cada elemento de los buckets ordenados en el arreglo original

Bucket Sort



Arreglo de datos a ordenar

Arreglo de buckets

Arreglo de datos ordenados

Quick sort

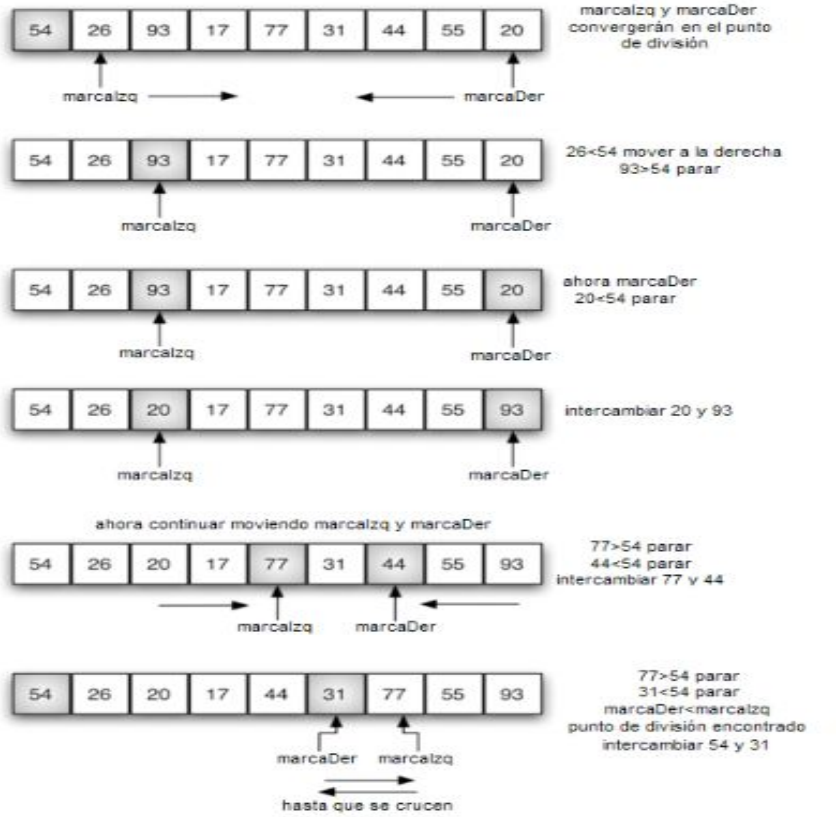
- Es un algoritmo clasificado del tipo divide y vencerás. Es la técnica de ordenamiento conocida más rápida
- Este algoritmo es simple en teoría, pero muy complicado de codificar. Por largos años los científicos de computación no lograban una implementación práctica de este algoritmo.

Quick sort

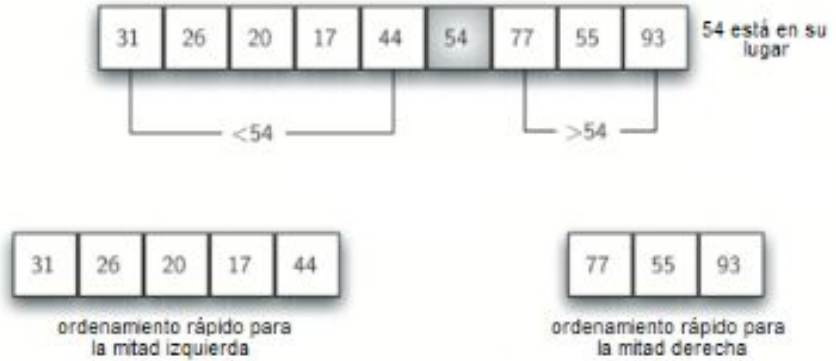
Pasos del algoritmo

- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**.
- **Resituar** los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán

Quick sort


54 26 93 17 77 31 44 55 20
marcalzq
marcaDer
marcalzq y marcaDer convergerán en el punto de división
54 26 93 17 77 31 44 55 20
marcalzq
marcaDer
26<54 mover a la derecha
93>54 parar
54 26 93 17 77 31 44 55 20
marcalzq
marcaDer
ahora marcaDer 20<54 parar
54 26 20 17 77 31 44 55 93
marcalzq
marcaDer
intercambiar 20 y 93
ahora continuar moviendo marcalzq y marcaDer
54 26 20 17 77 31 44 55 93
marcalzq
marcaDer
77>54 parar
44<54 parar
intercambiar 77 y 44
54 26 20 17 44 31 77 55 93
marcaDer
marcalzq
hasta que se crucen
77>54 parar
31<54 parar
marcaDer<marcalzq
punto de división encontrado
intercambiar 54 y 31

Quick sort


31 26 20 17 44 54 77 55 93
54 está en su lugar
<54
>54
31 26 20 17 44
ordenamiento rápido para la mitad izquierda
77 55 93
ordenamiento rápido para la mitad derecha

Quick sort

```
public class QuickSort {
    public static void q_sort(int[] arreglo, int izq, int
der) {
        int pivot, guarda_izq, guarda_der;
        guarda_izq = izq;
        guarda_der = der;
        pivot = arreglo[izq];
        while (izq < der) {
            while ((arreglo[der] >= pivot) && (izq < der))
                der--;
            if (izq != der) {
                arreglo[izq] = arreglo[der];
                izq++;          }
            while ((arreglo[izq] <= pivot) && (izq < der))
                izq++;
            if (izq != der) {
                arreglo[der] = arreglo[izq];
                der--; } }
        arreglo[izq] = pivot;
        pivot = izq;
        izq = guarda_izq;
        der = guarda_der;
        if (izq < pivot)
            q_sort(arreglo, izq, pivot - 1);
        if (der > pivot)
            q_sort(arreglo, pivot + 1, der);
    }
```

Quick sort

- Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.
- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño.
- En el peor caso, el pivote termina en un extremo de la lista. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas.
- No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.
- Dado que la eficiencia depende de la elección del pivote, se dice que el quicksort es un algoritmo inestable

Algoritmos de búsqueda

- Otra tema fundamental en el trabajo con estructuras de datos es la **búsqueda** de un elemento en la misma
- Es frecuente contar con grandes cantidades de datos guardados en un arreglo y necesitamos determinar si un elemento se encuentra en el arreglo. A este proceso se le llama búsqueda.
- Si el arreglo se encuentra desordenado la única forma de realizar la búsqueda es recorriéndolo.
- Este método de búsqueda se denomina secuencial.
- Búsqueda Secuencial o lineal: consiste en recorrer el arreglo comparando el elemento buscado con el elemento actual.

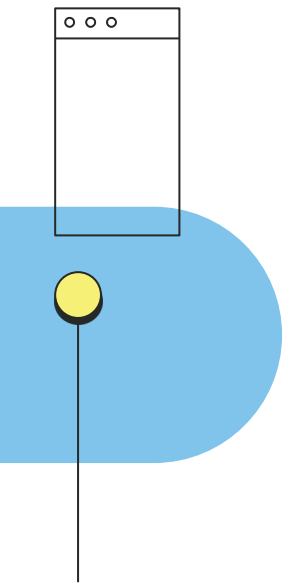
Algoritmos de búsqueda

Cuando el arreglo se encuentra ordenado el método de búsqueda más eficiente es la búsqueda binaria.

La algoritmo de búsqueda binaria después de cada comparación elimina la mitad de los elementos del arreglo en el que busca.

El algoritmo localiza el elemento de la mitad y lo compara, si es el elemento buscado, finaliza la búsqueda de lo contrario si es mayor se queda con la mitad derecha del arreglo y si es menor con la mitad izquierda del mismo.

¿Cómo harían para implementar la búsqueda binaria?



Ordenamiento de objetos

¿Consideran que requiere un tratamiento particular?

Ordenamiento de objetos

Los diversos algoritmos que se hemos estudiado siempre han ordenado arrays de un tipo de dato simple: int, double, ...
La clase Vector de Java está diseñada para almacenar objetos de cualquier tipo.
Los elementos de un Vector son objetos de cualquier tipo (Object), para ser mas precisos, referencias a objetos.
Ordenar un vector implica, posiblemente, cambiar el orden que ocupan los objetos, según el **criterio de clasificación** de éstos. Este criterio debe permitir **comparar dos objetos** y determinar si un objeto es mayor, es menor o igual que otro.
Un Vector ordenado, w, posee las mismas propiedades de un array de tipo simple ordenado: si i,j son dos enteros cualesquiera en el rango 0 .. w.size()-1, siendo i < j, entonces w.elementAt(i) es menor o igual que w.elementAt(j).

Ordenamiento de objetos

Para comparar objetos necesitamos saber que criterio seguir para determinar que el objeto p1 es menor que el objeto p2.
Una alternativa consiste en declarar una interface con los métodos menorQue(), menorIgualQue() …, y que las clases de los objetos que se ordenan implementen la interface.

```
interface Comparador
{
boolean igualQue(Object op2); boolean menorQue(Object op2); boolean menorIgualQue(Object op2); boolean mayorQue(Object op2); boolean mayorIgualQue(Object op2);
}
```

Es responsabilidad de la clase que implementa Comparador definir el criterio que aplica para menor o mayor.