Algoritmos de Ordenamiento

Jaime Torres Oquillas

INDICE

1 - INTRODUCCION

Los algoritmos de ordenamiento nos permite, como su nombre lo dice, ordenar. En este caso, nos serviran para ordenar vectores o matrices con valores asignados aleatoriamente. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y revisando el código, escrito en Java, de cada algoritmo.

Este informe nos permitirá conocer mas a fondo cada metodo distinto de ordenamiento, desde uno simple hasta el mas complejo. Se realizaran comparaciones en tiempo de ejecucion, pre-requisitos de cada algoritmo, funcionalidad, alcance, etc.

2 – TIPOS DE ALGORITMOS

Para poder ordenar una cantidad determinada de numeros almacenadas en un vector o matriz, existen distintos metodos (*algoritmos*) con distintas caracteristicas y complejidad.

Existe desde el metodo mas simple, como el *Bubblesort* (o Método Burbúja), que son simples iteraciones, hasta el *Quicksort* (Método Rápido), que al estar optimizado usando recursion, su tiempo de ejecucion es menor y es más efectivo.

2.1 – METODOS ITERATIVOS

Estos metodos son simples de entender y de programar ya que son iterativos, simples ciclos y sentencias que hacen que el vector pueda ser ordenado.

Dentro de los Algoritmos iterativos encontramos:

- Burbuja
- Inserción
- Selección
- Shellsort

2.2 - METODOS RECURSIVOS

Estos metodos son aún mas complejos, requieren de mayor atención y conocimiento para ser entendidos. Son rápidos y efectivos, utilizan generalmente la técnica *Divide y vencerás*, que consiste en dividir un problema grande en varios pequeños para que sea más fácil resolverlos.
Mediante llamadas recursivas a si mismos, es posible que el tiempo de ejecución y de ordenación sea más optimo.

Dento de los algoritmos recursivos encontramos:

- Ordenamiento por Mezclas (*merge*)
- Ordenamiento Rápido (*quick*)

3 – METODO DE LA BURBUJA

El metodo de la burbuja es uno de los mas simples, es tan facil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.

Por ejemeplo, imaginemos que tenemos los siguientes valores:

5	6	1	0	3

Lo que haria una burbuja simple, seria comenzar recorriendo los valores de izq. a derecha, comenzando por el 5. Lo compara con el 6, con el 1, con el 0 y con el 3, si es mayor o menor (dependiendo si el orden es ascendiente o descendiente) se intercambian de posicion. Luego continua con el siguiente, con el 6, y lo compara con todos los elementos de la lista, esperando ver si se cumple o no la misma condicion que con el primer elemento. Asi, sucesivamente, hasta el ultimo elemento de la lista.

3.1 – BURBUJA SIMPLE

Como lo describimos en el *item* anterior, la burbuja mas simple de todas es la que compara todos con todos, generando comparaciones extras, por ejemplo, no tiene sentido que se compare con sigo mismo o que se compare con los valores anteriores a el, ya que supuestamente, ya estan ordenados.

```
for (i=1; i<LIMITE; i++)
    for j=0 ; j<LIMITE - 1; j++)
          if (vector[j] > vector[j+1])
               temp = vector[j];
               vector[j] = vector[j+1];
               vector[j+1] = temp;
```

3. 2- BURBUJA MEJORADA

Una nueva version del metodo de la burbuja seria limitando el numero de comparaciones, dijimos que era inutil que se compare consigo misma. Si tenemos una lista de 10.000 elementos, entonces son 10.000 comparaciones que estan sobrando. Imaginemos si tenemos 1.000.000 de elementos. El metodo seria mucho mas optimo con “n” comparaciones menos (n = total de elementos).

3. 3- BURBUJA OPTIMIZADA

Si al cambio anterior (el de la burbuja mejorada) le sumamos otro cambio, el hecho que los elementos que estan detrás del que se esta comparando, ya estan ordenados, las comparaciones serian aun menos y el metodo seria aun mas efectivo.
Si tenemos una lista de 10 elementos y estamos analizando el quinto elemento, que sentido tiene que el quinto se compare con el primero, el segundo o el tercero, si supuestamente, ya estan ordenados? Entonces optimizamos mas aun el algoritmo, quedando nuestra version final del algoritmo optimizado de la siguiente manera:

```
Bubblesort(int matriz[])
{
        int buffer;
        int i,j;
        for(i = 0; i < matriz.length; i++)
        {
                for(j = 0; j < i; j++)
                {
                        if(matriz[i] < matriz[j])
                        {
                                buffer = matriz[j];
                                matriz[j] = matriz[i];
                                matriz[i] = buffer;
                        }
                }
        }
}
```

4 – INSERCION Y SELECCION

```
Insercion(int matrix[])
{
      int i, temp, j;
      for (i = 1; i < matrix.length; i++)
      {
            temp = matrix[i];
            j = i - 1;
            while ( (matrix[j] > temp) && (j >= 0) )
            {
                  matrix[j + 1] = matrix[j];
                  j--;
            }
            matrix[j + 1] = temp;
      }
}
Seleccion(int[]matrix)
{
      int i, j, k, p, buffer, limit = matrix.length-1;
      for(k = 0; k < limit; k++)
      {
            p = k;
            for(i = k+1; i <= limit; i++)
                  if(matrix[i] < matrix[p]) p = i;
            if(p != k)
            {
                  buffer = matrix[p];
                  matrix[p] = matrix[k];
                  matrix[k] = buffer;
            }
      }
}
```

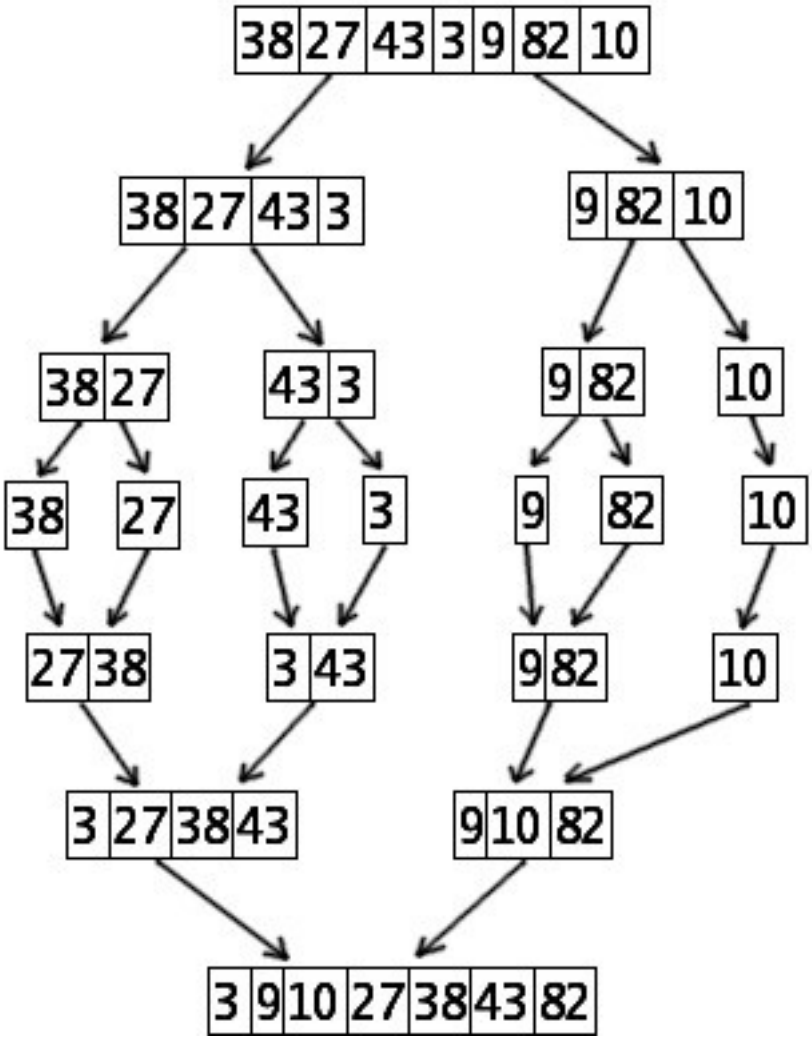
El bucle principal de la ordenacion por insercion va examinando sucesivamente todos los elementos de la matriz desde el segundo hasta el n-ésimo, e inserta cada uno en el lugar adecuado entre sus precedesores dentro de la matriz.

La ordenacion por selección funciona seleccionando el menor elemento de la matriz y llevandolo al principio; a continuacion selecciona el siguiente menor y lo pone en la segunda posicion de la matrizm y asi sucesivamente.

5 – ORDENAMIENTO POR MEZCLA

Este algoritmo consiste basicamente en dividir en partes iguales la lista de numeros y luego mezclarlos comparandolos, dejandolos ordenados.
Si se piensa en este algoritmo recursivamente, podemos imaginar que dividirá la lista hasta tener un elemento en cada lista, luego lo compara con el que está a su lado y según corresponda, lo situa donde corresponde.

En la siguiente figura podemos ver como funciona:



El algoritmo de ordenamiento por mezcla (*Mergesort*) se divide en dos procesos, primero se divide en partes iguales la lista:

```
public static void mergesort(int[ ] matrix, int init, int n)
{
      int n1;
      int n2;
      if (n > 1)
      {
            n1 = n / 2;
            n2 = n - n1;
            mergesort(matrix, init, n1);
            mergesort(matrix, init + n1, n2);
            merge(matrix, init, n1, n2);
      }
}
```

Y el algoritmo que nos permite mezclar los elementos según corresponda:

```
private static void merge(int[ ] matrix, int init, int n1, int n2)
{
      int[ ] buffer = new int[n1+n2];
      int temp  = 0;
      int temp1 = 0;
      int temp2 = 0;
      int i;
      while ((temp1 < n1) && (temp2 < n2))
      {
            if (matrix[init + temp1] < matrix[init + n1 + temp2])
                  buffer[temp++] = matrix[init + (temp1++)];
            else
                  buffer[temp++] = matrix[init + n1 + (temp2++)];
      }
      while (temp1 < n1)
            buffer[temp++] = matrix[init + (temp1++)];
      while (temp2 < n2)
            buffer[temp++] = matrix[init + n1 + (temp2++)];
      for (i = 0; i < n1+n2; i++)
            matrix[init + i] = buffer[i];
}
```

6 – METODO SHELLSORT

Este metodo es una mejora del algoritmo de ordenamiento por Insercion (*Insertsort*). Si tenemos en cuenta que el ordenamiento por insercion es mucho mas eficiente si nuestra lista de numeros esta semi-ordenada y que desplaza un valor una unica posicion a la vez.
Durante la ejecucion de este algoritmo, los numeros de la lista se van casi-ordenando y finalmente, el ultimo paso o funcion de este algoritmo es un simple metodo por insercion que, al estar casi-ordenados los numeros, es más eficiente.

El algoritmo:

```
public void shellSort(int[] matrix)
{
      for ( int increment = matrix.length / 2; increment > 0; increment =
      (increment == 2 ? 1 : (int) Math.round(increment / 2.2)))
      {
          for (int i = increment; i < matrix.length; i++)
        {
               for (int j = i; j >= increment && matrix[j - increment] >
                matrix[j]; j -= increment)
             {
                  int temp = matrix[j];
                  matrix[j] = matrix[j - increment];
                  matrix[j - increment] = temp;
               }
          }
      }
}
```

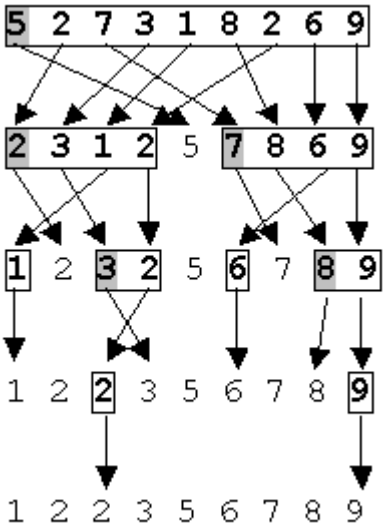
7 - METODO RAPIDO (*quicksort*)

Sin duda, este algoritmo es uno de los mas eficientes. Este metodo es el mas rapido gracias a sus llamadas recursivas, basandose en la teoria de *divide y vencerás.*

Lo que hace este algoritmo es dividir recurvisamente el vector en partes iguales, indicando un elemento de inicio, fin y un pivote (o comodin) que nos permitira segmentar nuestra lista. Una vez dividida, lo que hace, es dejar todos los mayores que el pivote a su derecha y todos los menores a su izq. Al finalizar el algoritmo, nuestros elementos estan ordenados.

Por ejemplo, si tenemos 3 5 4 8 basicamente lo que hace el algoritmo es dividir la lista de 4 elementos en partes iguales, por un lado 3, por otro lado 4 8 y como comodin o pivote el 5. Luego pregunta, 3 es mayor o menor que el comodin? Es menor, entonces lo deja al lado izq. Y como se acabaron los elementos de ese lado, vamos al otro lado. 4 Es mayor o menor que el pivote? Menor, entonces lo tira a su izq. Luego pregunta por el 8, al ser mayor lo deja donde esta, quedando algo asi:
3 4 5 8

En esta figura se ilustra de mejor manera un vector con mas elementos, usando como pivote el primer elemento:



El algoritmo es el siguiente:

```
public void _Quicksort(int matrix[], int a, int b)
{
	this.matrix = new int[matrix.length];
	int buf;
	int from = a;
	int to = b;
	int pivot = matrix[(from+to)/2];
	do
	{
		while(matrix[from] < pivot)
		{
			from++;
		}
		while(matrix[to] > pivot)
		{
			to--;
		}
		if(from <= to)
		{
			buf = matrix[from];
			matrix[from] = matrix[to];
			matrix[to] = buf;
			from++; to--;
		}
	}while(from <= to);
	if(a < to)
	{
		_Quicksort(matrix, a, to);
	}
	if(from < b)
	{
		_Quicksort(matrix, from, b);
	}
	this.matrix = matrix;
}
```

8 – COMPLEJIDAD

Cada algoritmo de ordenamiento por definicion tiene operaciones y calculos minimos y maximos que realiza (complejidad), a continuacion una tabla que indica la cantidad de calculos que corresponden a cada metodo de ordenamiento:

Algoritmo	Operaciones máximas
Burbuja	$\Omega(n^2)$
Insercion	$\Omega(n^2/4)$
Selección	$\Omega(n^2)$
Shell	$\Omega(n \log^2 n)$
Merge	$\Omega(n \log n)$
Quick	$\Omega(n^2)$ en peor de los casos y $\Omega(n \log n)$ en el promedio de los casos.

9 – COMPARACION DE TIEMPOS

Se han ordenado una cantidad determinada de elementos aleatorios en una lista mediante distintos metodos de ordenamiento. (en segundos)

256 elementos		512 elementos	
Burbuja:	0.0040	Burbuja:	0.0050
Seleccion:	0.0030	Seleccion:	0.0040
Insercion:	0.0040	Insercion:	0.0050
Rapido:	0.0010	Rapido:	0.0010
Shell:	0.0010	Shell:	0.0020
Merge:	0.0040	Merge:	0.0030

2048 elementos		16384 elementos	
Burbuja:	0.022	Burbuja:	1.055
Seleccion:	0.015	Seleccion:	0.9
Insercion:	0.013	Insercion:	0.577
Rapido:	0.0010	Rapido:	0.0080
Shell:	0.0060	Shell:	0.0090
Merge:	0.0050	Merge:	0.014

262144 elementos		2097152 elementos	
Burbuja:	178.205	Burbuja:	11978.107
Seleccion:	158.259	Seleccion:	10711.01
Insercion:	94.461	Insercion:	7371.727
Rapido:	0.061	Rapido:	0.596
Shell:	0.086	Shell:	0.853
Merge:	0.201	Merge:	1.327

Como podemos analizar, el algoritmo que se va demorando cada vez mas tiempo es el de la burbuja, luego de selección y tercero el insercion. Los algoritmos que los siguen son el Shell y el de ordenacion por mezcla, pero el más optimo es el “Rapido”