Métodos de ordenación (*Sorting*)

Generalidades

- Los métodos de ordenación hacen parte de la gran mayoría de sistemas transaccionales, de generación de reportes y gestores de base de datos.
- La idea básica es que los datos a ordenar poseen una llave y se desea ordenarlos bien sea en orden ascendente ó descendente con respecto a la llave.

Características generales

- Todos los métodos de ordenación dependen de dos operaciones básicas:
  - Comparar un par de llaves, para establecer si una es menor (o no) que la otra. (Operación *less()*)
  - Intercambiar la posición de un par de elementos. (Operación *exch()*)

Modelo general para métodos de ordenamiento

Los distintos métodos se implementan como bibliotecas de funciones con el siguiente patrón:

class OrdenacionNN		
static void	sort(Comparable[] a)	\\ Ordenar el arreglo
static int	less(Comparable a, Comparable b)	\\ Determinar si a<b
static void	exch(Comparable[] a, int i, int j)	\\ Intercambiar posiciones i y j
static void	show(Comparable[] a)	\\ Imprimir el arreglo
static boolean	isSorted(Comparable[] a)	\\ Verificar si está en orden

Modelo del problema

- isSorted() permite verificar si el vector está en orden.
- El tiempo de ejecución lo determinan principalmente las instrucciones de comparación e intercambio. Se toman estas como modelo de costo.
- Uso de memoria. Algunos algoritmos operan sobre los datos en el mismo vector de entrada (*in-situ*). Otros requieren copiar los datos, lo cual conlleva un consumo adicional.

Tipos de datos

- Se generaliza a cualquier tipo que posea una relación de orden. Esta relación define un orden total: Reflexiva, antisimétrica, transitiva.
- La generalización se logra estableciendo que implementan la interfaz Comparable al declarar el tipo.

`x.compareTo(y)` devuelve:
-1 si x es menor a y
0 si x es igual a y
1 si x es mayor que y

Ordenacion por selección

- Idea general:
  - Buscar el menor dato e intercambiarlo con la posición 0 del arreglo.
  - Repetir buscando el menor entre las posiciones i..N e intercambiando con la posición i.
  - Al completar la iteración N-1, el vector ya se encuentra ordenado.

Ordenación por selección

```
public static void sort(Comparable[] a) {
        int N = a.length;
        for (int i = 0; i < N-1; i++) {
            int min = i;
            for (int j = i+1; j < N; j++) {
                if (less(a[j], a[min])) min = j;
            }
            exch(a, i, min);
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }
```

Ver código fuente completo

Análisis

Proposición

- El número de comparaciones es ~$N^2/2$.
- El número de intercambios es ~N.

Observaciones:

- El tiempo de ejecución es independiente de las entradas.
- El número de intercambios es muy pequeño.

Ordenación por Inserción

Idea general:

- Empezando de la posición 1 hasta la N:
- Si el elemento actual es menor a su predecesor, se intercambian. Se continua hasta no encontrar un elemento menor.

Ordenación por Inserción

```
public static void sort(Comparable[] a) {
        int N = a.length;
        for (int i = 1; i < N; i++) {
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
                exch(a, j, j-1);
            }
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }
```

Ver código fuente completo

Observaciones

- El tiempo de ejecución es dependiente de la entrada particular.
- Es menor para entradas parcialmente ordenadas.

Análisis

Proposición

- En promedio se hacen ~$N^2/4$ comparaciones y ~$N^2/4$ intercambios.
- En el peor caso se hacen ~$N^2/2$ comparaciones y ~$N^2/2$ intercambios.
- En un mejor caso se hacen N-1 comparaciones y 0 intercambios.

Generalidades sobre la ordenación

- Se llama ***inversión*** a un par de datos fuera de orden.
- Se dice que el vector está parcialmente ordenado si el número de inversiones es menor a un factor constante del tamaño del arreglo.
- Ejemplo: El String E J E M P L O tiene 4 inversiones: J-E, M-L, P-L, P-O

Caso de la ordenación por inserción

Proposición

- El número de intercambios realizados por el algoritmo de inserción es igual al número de inversiones del arreglo.
- El número de comparaciones es igual al número de inversiones, más N-1.

Shellsort

- Extiende el método de InsertSort, pero operando no entre elementos adyacentes, sino con una secuencia de saltos (*gaps*) predeterminada.
- Dado un tamaño de salto h, al final de cada iteración toda subsecuencia de saltos de tamaño h está ordenada.
- Posteriormente se reduce h, hasta llegar a 1, momento en el que es equivalente a un InsertSort, pero reduciendo significativamente el número de intercambios a realizar.

Secuencia de saltos

- Hay muchas secuencias posibles, pero no hay una demostrada óptima.
- Una secuencia que arroja muy buen desempeño es

$$h=\frac{1}{2}\left(3^k-1\right),\ \ k=1,2,\ldots$$

empezando en el menor valor de h mayor o igual a $\lfloor N/3 \rfloor$

Algoritmo Shellsort

```
public static void sort(Comparable[] a) {
        int N = a.length;
        // 3x+1 increment sequence:  1, 4, 13, 40, 121, 364, 1093, ...
        int h = 1;
        while (h < N/3) h = 3*h + 1;
        while (h >= 1) {
            // h-sort the array
            for (int i = h; i < N; i++) {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h) {
                    exch(a, j, j-h);
                }
            }
            assert isHsorted(a, h);
            h /= 3;
        }
        assert isSorted(a);
    }
```

Ver código fuente completo

Eficiencia de Shellsort

- Al usar saltos grandes (h>1) se logra que los elementos se acerquen a su posición definitiva en pocos pasos.
- Cuando h=1, se hace la misma secuencia de InsertSort, pero en general se requeriran pocos intercambios.
- No hay una solución análitica para la eficiencia promedio. El número de comparaciones de esta versión es $\sim N^{3/2}$ (que es mucho menor a N^2).