Métodos de ordenación

Quicksort
(Ordenación rápida)

Generalidades

- Es una de las técnicas de ordenación más utilizadas
  - Muy eficiente en el caso medio
  - Tiene unas constantes asociadas pequeñas
  - Opera *in-situ*

Estrategia de Quicksort

- También se trata de un algoritmo divide y vencerás.
- Su estrategia es la opuesta a `mergesort`: Primero se hace el trabajo en el rango completo y luego se invoca recursivamente en las divisiones.
- A diferencia de `mergesort`, las dos particiones del rango inicial no necesariamente son de igual tamaño.

Operación sobre un rango: Particionamiento

- Se toma un elemento arbitrario como pivote y se invoca el algoritmo `partition` en el rango `lo..hi`. Este intercambia los elementos del vector de forma que se cumplen las siguientes postcondiciones:
  - `a[k]` queda en la posición final para el pivote (algún elemento k del rango).
  - `a[i]≤a[k]` para todo `lo≤i<k`.
  - `a[k]≤a[j]` para todo `k<j≤hi`.
- Se invoca recursivamente en los intervalos `lo..k-1` y `k+1..hi`.

Implementación del algoritmo de particion

```
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    Comparable v = a[lo];

    while (true) {

        while (less(a[++i], v))
            if (i == hi) break;

        while (less(v, a[--j]))
            if (j == lo) break;

        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

Eficiencia del algoritmo `partition`

- Dado un rango de N=hi-lo+1 elementos:
- El pivote v se compara una vez con cada uno de los N-1 elementos restantes.
- Los ciclos `while` internos hacen una comparación adicional cuando no se cumple, o sea que se tienen 2 comparaciones más.
- En total se realizan N+1 comparaciones en un rango de tamaño N.

Implementación de quicksort

```
public static void sort(Comparable[] a) {
    StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
    assert isSorted(a);
}


// quicksort the subarray from a[lo] to a[hi]
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
    assert isSorted(a, lo, hi);
}
```

Eficiencia de Quicksort

- Es dependiente de la entrada. A diferencia de mergesort, el vector de entrada no se divide en mitades iguales.
- En el caso ideal cuando el vector se divide a la mitad, el número de comparaciones lo describe la recurrencia:

$$C(N) = 2C(N/2) + N$$

que es exactamente la misma recurrencia de mergesort, con solución:

$$N \lg(N)$$

Eficiencia media de Quicksort

- En general se debe considerar la posibilidad de que el pivote quede ubicado en cualquiera de las posiciones del arreglo.
- Se toma el número promedio de comparaciones de todos los posibles casos y se suma el número de comparaciones de la invocación inicial, dando la recurrencia

$$C(N) = (N+1) + \frac{C(0) + \ldots + C(N-1)}{N} + \frac{C(N-1) + \ldots + C(0)}{N}$$

Solución de la recurrencia (1)

- Se observa que C(0)=C(1)=0.
- Tomando la diferencia de 2 casos consecutivos

$$\begin{aligned} NC(N) &= N(N+1) + 2\left(C(0) + C(1) + +C(N-1)\right) \\ (N-1)C(N-1) &= (N-1)N + 2\left(C(0) + C(1) + +C(N-2)\right) \\ \hline NC(N) - (N-1)C(N-1) &= 2N + 2C(N-1) \end{aligned}$$

- Agrupando y simplificando

$$\begin{aligned} NC(N) &= 2N + (N+1)C(N-1) \\ \frac{C(N)}{N+1} &= \frac{2}{N+1} + \frac{C(N-1)}{N} \\ &\vdots \\ \frac{C(2)}{3} &= \frac{2}{3} + \frac{C(1)}{2} \\ \frac{C(1)}{2} &= 0 \end{aligned}$$

Solución de la recurrencia (2)

- Y sustituyendo

$$\frac{C(N)}{N+1} = 2\left(\frac{1}{N+1} + \frac{1}{N} + \ldots + \frac{1}{3}\right)$$

- donde aparece la serie armónica

$$\sum_{k=1}^{n} \frac{1}{k} \sim \ln n$$

- permitiendo concluir

$$C(N) \sim 2N \ln N$$

Referencia: La serie armónica

Peor caso de Quicksort

- Se presenta cuando el pivote queda ubicado en uno de los extremos del intervalo en todas las llamadas recursivas.
- Esto lleva a un número de comparaciones de la forma:

$$C(N) = N + (N-1) + \ldots + 1 = \frac{N(N+1)}{2}$$

que es cuadrática. Sin embargo solo ocurre en casos muy específicos. Aleatorizar el vector antes de realizar la ordenación hace muy improbable que se presenten estos casos.

Mejoras a la implementación

- Algoritmo híbrido: Es más eficiente utilizar ordenación por inserción para vectores muy pequeños (*e.g.* N<15).
- Media de tres: Para evitar que el pivote sea un valor extremo, tomar 3 muestras en el intervalo `lo..hi` y utilizar el valor medio de las tres.
- Cuando el vector posee muchas llaves repetidas, conviene dividir el vector en tres partes: Menores al pivote, repeticiones del pivote y mayores al pivote. (Ver Quick3way)