

TAREA PRACTICA P2 IKER NUÑEZ MEANA UO306425

Tabla 1: Algoritmo de ordenación Burbuja

Para n = 1 en los tres casos y sin optimización:

n	Ordenado	Inverso	Aleatorio
10000	418	1096	851
20000	1733	4372	3515
40000	6924	17544	14080
80000	27423	69033	55686
160000	109910	276233	222360
320000	fdt	fdt	fdt
640000	fdt	fdt	fdt
1280000	fdt	fdt	fdt

¿Son los tiempos obtenidos los esperados?

Para razonarlo, nos apoyamos en la implementación:

```
/* Ordenacion por el metodo de Burbuja */
public static void burbuja(int[] a) {
    int n = a.length;
    int x;
    for (int i = 0; i <= n - 2; i++)
        for (int j = n - 1; j > i; j--)
            if (a[j - 1] > a[j]) {
                // intercambio de valores
                x = a[j - 1];
                a[j - 1] = a[j];
                a[j] = x;
            }
}
```

En este algoritmo, el array de entrada no es muy notable en el tiempo que tarde, porque en todos los casos recorre el array entero por pares, por los dos for's anidados, luego la diferencia estará en todos los casos en el número de veces que entra por el if / no.

En el caso del array ordenado, no entrara nunca por el if por eso es el más rápido de todos, sin embargo, no es mucho más rápido por lo comentado. Si el array de entrada esta ordenado de forma aleatoria, algunas veces entrará por el if y otras no, por eso tarda más que el ordenado.

Para el último caso, un array ordenado inversamente, entrará por el if en todos los casos, por eso es el más lento, pero no por mucho.

Tabla 2: Algoritmo de ordenación Selección

Para $n = 1$ en los tres casos y sin optimización:

n	Ordenado	Inverso	Aleatorio
10000	395	349	396
20000	1614	1380	1579
40000	6372	5625	6337
80000	25329	22125	25334
160000	100812	88357	100603
320000	fdt	fdt	fdt
640000	fdt	fdt	fdt
1280000	fdt	fdt	fdt

¿Son los tiempos obtenidos los esperados?

Para razonarlo, nos apoyamos en la implementación:

```
/* Ordenacion por el metodo de Seleccion */
public static void seleccion(int[] a) {
    int n = a.length;
    int x;
    int posmin;
    for (int i = 0; i < n - 1; i++) { // Buscar la posicion del mas
        // pequeno de los que quedan
        posmin = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[posmin])
                posmin = j;
        // Intercambia el que toca con el mas pequeo
        x = a[posmin];
        a[posmin] = a[i];
        a[i] = x;
    } // for
}
```

En este caso, lo que hace es para cada i busca el número más bajo desde $i + 1$ hasta el final y, si este número es más bajo que el que está en la posición i , intercambia su posición este.

Luego, para un array ordenado, recorrerá todo el array $n \cdot n$ veces por los dos for's anidados y no hará modificación alguna ya que nunca encontrará un número más bajo por el que intercambiar la posición.

Para un array aleatorio, la diferencia en tiempo es poco notable, ya que algunas veces hará modificaciones y otras no, pero recorrerá el array el mismo numero de veces.

Por último, para un array ordenado inversamente, se dan los mejores tiempos ya que hará modificaciones constantes pero rápidas e ira ordenando el array. Es por esto que los tiempos en este caso son los más bajos.

Tabla 3: Algoritmo de ordenación Inserción

Para $n = 1$ en los tres casos y sin optimización:

n	Ordenado	Inverso	Aleatorio
10000	0	683	338
20000	1	2698	1359
40000	1	10774	5382
80000	2	43098	21628
160000	3	172258	86641
320000	6 fdt		356737
640000	12 fdt	fdt	fdt
1280000	24 fdt	fdt	fdt

¿Son los tiempos obtenidos los esperados?

Para razonarlo, nos apoyamos en la implementación:

```
/* Ordenacion por el metodo de Insercion */
public static void insercion(int[] a) {
    int n = a.length;
    for (int i = 1; i <= n - 1; i++) {
        int x = a[i];
        int j = i - 1;
        while (j >= 0 && x < a[j]) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = x;
    } // for
}
```

Lo que hace el algoritmo de inserción es iterar desde el primer elemento y crear 2 particiones, una que será la ordenada (a su izquierda) y otra que es la que está sin ordenar (a su derecha).

Para un array de entrada ordenado realmente no hace nada, solo recorre el array pero el resto son operaciones $O(1)$ ya que en el while no entra nunca al no ser x

$< a[j]$ (es decir x menor que el anterior). Luego es por eso que los tiempos salen tan pequeños.

Para el caso de un array aleatorio, algunas veces entrara en el while y otras no, por eso tarda mas que en el caso del array ordenado.

Por último, el array ordenado inversamente es el peor caso en cuanto a tiempos, ya que en todos los casos va a entrar en el while y de hecho va a tener que ir moviendo todas las posiciones a la izquierda por el hecho de que este ordenado inversamente. Por ello, es el peor caso en cuanto a tiempos con diferencia.

Tabla 4: Algoritmo de ordenación Rápido

Para $n = 1$ en los tres casos y sin optimización:

n	Ordenado	Inverso	Aleatorio
10000	4	4	4
20000	6	7	9
40000	12	13	17
80000	24	26	35
160000	51	61	75
320000	112	129	169
640000	240	254	343
1280000	493	528	710

¿Son los tiempos obtenidos los esperados?

Para razonarlo, nos apoyamos en la implementación:

```
public static void rapido(int[] v) {
    rapirec(v, iz: 0, v.length - 1);
}

private static void rapirec(int[] v, int iz, int de) {
    int m;
    if (de > iz) {
        m = particion(v, iz, de);
        rapirec(v, iz, m - 1);
        rapirec(v, m + 1, de);
    }
}
```

```

private static void intercambiar(int[] v, int i, int j) {
    int t;
    t = v[i];
    v[i] = v[j];
    v[j] = t;
}

/**
 * Deja el pivote en una posicion tal que a su izquierda no
 * hay ningun mayor, ni a la derecha ningun menor.
 * Es un proceso lineal O(n).
 */
private static int particion(int[] v, int iz, int de) {
    int i, pivote;
    intercambiar(v, (iz + de) / 2, iz);
    // el pivote es el de centro y se cambia con el primero
    pivote = v[iz];
    i = iz;
    for (int s = iz + 1; s <= de; s++)
        if (v[s] <= pivote) {
            i++;
            intercambiar(v, i, s);
        }
    intercambiar(v, iz, i); // se restituye el pivote donde debe estar
    return i; // retorna la posicion en que queda el pivote
}

```

En este caso, nos encontramos con el algoritmo de QuickSort, que es una implementación de un algoritmo de tipo Divide y Vencerás. En este caso, se escoge un pivote que es el primer elemento de la partición y lo que se hace es ir haciendo el problema mas pequeño hasta que resulte una operación simple.

En este caso, vemos que el array de entrada no afecta demasiado a los tiempos de ejecución, este ordenado o no. Esto se debe a que simplemente no mira cada par de elementos, sino solo compara con el pivote escogido tras hacer las particiones. Los tiempos salen relativamente parecidos en todos los casos, pero vemos que tiene sentido que lo sean.

A partir de las complejidades y de los datos anteriores, ¿cuántos días tardaría cada algoritmo en ordenar un vector de 81.920.000 para un array ordenado de forma aleatoria?

Algoritmo de Burbuja:

Datos: $n_1 = 160000$, $t_1 = 222360$ ms

$n_2 = 81.920.000$, $t_2 = ?$

Complejidad del algoritmo: $O(n^2)$ en todos los casos (pdf teoría)

$t_2 = 222360 * ((81.920.000)^2 / (160000)^2) = 5,82903 * 10^{10}$ ms

1 día -> 86.400.000 ms

Luego,

$$5,82903 * 10^{10} * 1 \text{ día} / 86.400.000 = 674,66 \text{ días} \rightarrow 675 \text{ días}$$

Algoritmo de Inserción:

Datos: $n_1 = 320000$, $t_1 = 356737 \text{ ms}$

$n_2 = 81.920.000$, $t_2 = ?$

Complejidad del algoritmo: $O(n^2)$ en este caso, que no es Caso Mejor (pdf teoría)

$$t_2 = 356737 * ((81.920.000)^2 / (320000)^2) = 2,33791 * 10^{10} \text{ ms}$$

1 día \rightarrow 86.400.000 ms

Luego,

$$2,33791 * 10^{10} \text{ ms} * 1 \text{ día} / 86.400.000 \text{ ms} = 279,59 \text{ días} \rightarrow 280 \text{ días}$$

Algoritmo de Selección:

Datos: $n_1 = 160000$, $t_1 = 100603 \text{ ms}$

$n_2 = 81.920.000$, $t_2 = ?$

Complejidad del algoritmo: $O(n^2)$ en todos los casos (pdf teoría)

$$t_2 = 100603 * ((81.920.000)^2 / (160000)^2) = 2,63724 * 10^{10} \text{ ms}$$

1 día \rightarrow 86.400.000 ms

Luego,

$$2,63724 * 10^{10} \text{ ms} * 1 \text{ día} / 86.400.000 \text{ ms} = 305,24 \text{ días} \rightarrow 306 \text{ días}$$

Algoritmo de QuickSorting:

Datos: $n_1 = 1280000$, $t_1 = 710 \text{ ms}$

$n_2 = 81.920.000$, $t_2 = ?$

Complejidad del algoritmo: $O(n \cdot \log(n))$, tiramos por el caso medio.

$$t_2 = 710 * ((81920000 * \log(81920000)) / (1280000 * \log(1280000))) = 58878,76732 \text{ ms}$$

1 día \rightarrow 86.400.000 ms

Luego,

$58878,76732\text{ms} * 1 \text{ día} / 86.400.000\text{ms} = 6,81466 * 10^{-4} \text{ días} \rightarrow \text{Menos de un día}$

Tabla 5: Rápido vs Inserción en tiempos bajos

Para $n = 100000$, sin optimización y orden aleatorio en ambos casos:

n	Rapido	Insercion	t rapido/t insercion
10	230	126	1,825396825
15	392	226	1,734513274
20	582	335	1,737313433
25	726	498	1,457831325
30	875	845	1,035502959
35	1068	1570	0,680254777
40	1589	1271	1,250196696
45	1900	2225	0,853932584
50	2301	2621	0,877909195
55	3055	2675	1,142056075
60	3487	3603	0,967804607
65	3179	3775	0,842119205
70	4358	5084	0,857199056
75	4960	5428	0,913780398
80	5289	5892	0,897657841
85	5599	6955	0,805032351
90	6195	7683	0,806325654
95	6265	6155	1,017871649
100	5802	6629	0,875245135

El algoritmo de QuickSort empieza a ser más rápido que el de inserción a partir de $n = 35$, para n 's más bajos el algoritmo de inserción es más rápido. Esto nos quiere decir que, a pesar de que el algoritmo de Quicksort sea el mejor algoritmo de ordenación de todos por complejidad, para tamaños pequeños no es significativo el algoritmo que uses en cuanto a tiempos de ejecución.

De forma opcional, se mandaba hacer un programa que según el tamaño del array eligiera el método de inserción o el método de Quicksort. En mi caso, Inserción era más rápido para arrays de tamaño ≤ 30 .

El programa principal sería este:

```
Run | Debug
public static void main(String arg[]) {
    long t1, t2;
    String opcion = arg[0];
    int repeticiones = Integer.parseInt(arg[1]);

    for (int n = 10; n <= 100; n += 5) {
        v = new int[n];
        long t = 0;

        for (int i = 1; i <= repeticiones; i++) {
            Vector.ordenAleatorio(v);

            t1 = System.currentTimeMillis();

            RapidoConInsercion.ordenar(v);

            t2 = System.currentTimeMillis();
            t = t + (t2 - t1);
        }

        System.out.println("n=" + n + "\tTiempo=" + t + " \tRepeticiones=" + repeticiones);
    }
}
```

Y la implementación del método ordenar sería la siguiente:

```
/* Ordenacion segun el tamaño del array de entrada */
public static void ordenar(int [] a){
    int n = a.length;

    //hasta n = 30, el algoritmo de insercion es mas rapido

    if(n <= 30){
        insercion(a);
    } else{
        rapido(a);
    }
}

/* Ordenacion por el metodo de Insercion */
private static void insercion(int[] a) {
    int n = a.length;
    for (int i = 1; i <= n - 1; i++) {
        int x = a[i];
        int j = i - 1;
        while (j >= 0 && x < a[j]) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = x;
    } // for
}

private static void rapido(int[] v) {
    rapirec(v, iz: 0, v.length - 1);
}
```

Si el array de entrada tiene un tamaño menor o igual que 30 llama al método de ordenar por Inserción, sino al de Quicksort.

Los tiempos resultantes para un orden aleatorio como el pedido y $n = 100000$ repeticiones (porque así podemos compararlos con los de la tabla 5 anterior):

n	t(ms)
10	109
15	218
20	350
25	556
30	1165
35	1798
40	2131
45	2606
50	2815
55	3258
60	3522
65	3908
70	4231
75	4618
80	5087
85	5553
90	5851
95	6000
100	6528

n	Rapido	Insercion	t rapido/t insercion
10	230	126	1,825396825
15	392	226	1,734513274
20	582	335	1,737313433
25	726	498	1,457831325
30	875	845	1,035502959
35	1068	1570	0,680254777
40	1589	1271	1,250196696
45	1900	2225	0,853932584
50	2301	2621	0,877909195
55	3055	2675	1,142056075
60	3487	3603	0,967804607
65	3179	3775	0,842119205
70	4358	5084	0,857199056
75	4960	5428	0,913780398
80	5289	5892	0,897657841
85	5599	6955	0,805032351
90	6195	7683	0,806325654
95	6265	6155	1,017871649
100	5802	6629	0,875245135

Como vemos, para n 's pequeños, efectivamente, los tiempos son mejores que en Quicksort y a partir de 30 los tiempos son mejores que en la otra tabla con Insercion. Luego, el algoritmo esta funcionando bien.