



Materia:

Estructura de datos

Tema:

“Métodos de ordenamiento”

Profesor:

Ray Parra

Horario:

13:00 pm a 14:00 pm

Alumno:

Perales Niebla Abner Jesús

Número de control:

17211551

Fecha de entrega:

30 - diciembre - 2018

Descripción del proyecto:

Desarrollar un programa que haga 30 pruebas de los métodos de ordenamiento vistos en clase, tomando el tiempo en cada prueba y evaluando cual es más rápido viendo el promedio.

A continuación se muestra el código del programa en grandes partes, pues ya está comentado.

```
② import java.util.ArrayList;
    import java.util.Scanner;

    public class ProyectoFinal_EstructuraDeDatos {

        /*
         * Proyecto final de Estructura de datos
         * Horario de la clase: Lunes a viernes, 13:00 pm a 14:00 pm
         * Alumno: Perales Niebla Abner Jesús
         * Profesor Ray Parra
         * Fecha de entrega en github: 30 diciembre 2018
         *
         * Instrucciones: hacer 30 pruebas con los 4 métodos de ordenamiento vistos en clase, ordenar el mismo
         * arreglo con los 4 métodos, pero un arreglo distinto por cada prueba. Comparar tiempos.
         */
        /*
         * Todos los siguientes son arreglos y variables globales necesarios para el proyecto
         */
        public int[] Arreglo;
        public int[] ArregloMerge;
        public int[] ArregloQuick;
        public int[] ArregloShell;
        public int[] ArregloBubble;

        public int longitud = 0;
        public ArrayList<Integer> izquierda = new ArrayList<>();
        public ArrayList<Integer> derecha = new ArrayList<>();

        //Variables de tiempo
        long BubbleS, Quicks, Shells, MergeS;
        long comienzo, fin, total;

        ② public static void main(String[] args) {
            // TODO Auto-generated method stub
            ProyectoFinal_EstructuraDeDatos PF = new ProyectoFinal_EstructuraDeDatos();
            PF.Instrucciones();
        }
    }
```

Para esta primera parte, vemos las librerías importadas para poder realizar varios procesos; así como también vemos todas las variables y arreglos usados para el programa. En el main se ejecuta el método Instrucciones() el cual realiza las pruebas.



```
/*
 * Este es el método en el que se ejecutaran todas las pruebas
 * Se presentan instrucciones generales y luego un ciclo de 30 iteraciones
 */
public void Instrucciones() {
    //Instrucciones generales
    P("Se hará la comparación con 30 arreglos.");
    P("Cada arreglo tendrá 10000 elementos aleatorios.");
    P("Cada arreglo se presenta al inicio de cada ciclo.");
    longitud = 5000;

    //Ciclo para las 30 pruebas
    for(int f = 0; f < 30; f++) {
        //Creamos el arreglo aleatorio y luego lo guardamos en los auxiliares
        Arreglo = Random(longitud);
        ArregloBubble = Arreglo;
        ArregloQuick = Arreglo;
        ArregloShell = Arreglo;
        ArregloMerge = Arreglo;

        P("Arreglo a ordenar.");
        Peek(Arreglo);
        /*
         * En cada módulo de cada método se realiza lo siguiente
         *
         * Se resetean las variables de tiempo a cero, para evitar que se sumen a otros tiempos.
         * Se toma el tiempo en milisegundos de inicio
         * Se realiza el proceso
         * Se toma el tiempo en milisegundos al fin del proceso
         * Se resta el final - el inicio y se obtiene el total
         * Se imprime el tiempo que duro y se suma el valor a una variable que obtiene todos los tiempos
         * por cada método.
         */
    }
}
```

En esta primera parte del método `Instrucciones()` se crea el arreglo aleatoriamente, con un método que se verá al final, y se le asigna ese arreglo a arreglos diseñados para cada uno de los métodos.

De esta forma cada método acomodará el mismo arreglo.



```
***** BubbleSort en ejecucion *****
comienzo = 0;
fin = 0;
total = 0;
comienzo = System.currentTimeMillis();
Bubble();
P("Burbuja Ordenado...");
fin = System.currentTimeMillis();
total = fin - comienzo;
P("BubbleSort: " + total);
BubbleS = BubbleS + total;
***** BubbleSort en ejecucion *****

***** QuickSort en ejecucion *****
comienzo = 0;
fin = 0;
total = 0;
comienzo = System.currentTimeMillis();
QuickSort(ArregloQuick, 0, longitud -1);
P("Quicksort Ordenado...");
fin = System.currentTimeMillis();
total = fin - comienzo;
P("QuickSort: " + total);
QuickS = QuickS + total;
***** QuickSort en ejecucion *****

***** ShellSort en ejecucion *****
comienzo = 0;
fin = 0;
total = 0;
comienzo = System.currentTimeMillis();
ShellSort(ArregloShell);
P("ShellSort Ordenado...");
fin = System.currentTimeMillis();
total = fin - comienzo;
P("ShellSort: " + total);
Shells = Shells + total;
***** ShellSort en ejecucion *****

***** MergeSort en ejecucion *****
comienzo = 0;
fin = 0;
total = 0;
comienzo = System.currentTimeMillis();
MergeSort();
P("MergeSort Ordenado...");
fin = System.currentTimeMillis();
total = fin - comienzo;
P("MergeSort: " + total);
MergeS = MergeS + total;
***** MergeSort en ejecucion *****
```

Aquí se ejecuta y se toma el tiempo de cada método, el cómo se hace se explica en el código en la página anterior.



```

//Guarda los tiempos de los metodos y el nombre
double[] metodos = {BubbleS/30, QuickS/30, Shells/30, MergeS/30};
String[] nombres = {"BubbleSort", "QuickSort", "ShellSort", "MergeSort"};
P("-----");
P("Promedio de los tiempos de ejecucion.");

//Con un mini burbuja se ordenan de menor a mayor junto al nombre
for(int e = 0; e < 3; e++) {
    for(int ee = 0; ee < 3; ee++) {
        if(metodos[ee] > metodos[ee + 1]){
            double ax = metodos[ee];
            metodos[ee] = metodos[ee + 1];
            metodos[ee + 1] = ax;

            String as = nombres[ee];
            nombres[ee] = nombres[ee + 1];
            nombres[ee + 1] = as;
        }
    }
}

//Se despliegan los tiempos de los metodos
for(int v = 0; v < 4; v++) {
    P(nombres[v] + ": " + metodos[v]);
}
P("Presione enter para salir.");
String enter = LeerS();
System.out.close();
}

```

Esta es la última parte de método Instrucciones(). Aquí se guardan los promedios de los tiempos en un arreglo y se acomodan del menor al mayor junto a su nombre, con el método burbuja.

A continuación presento los 4 métodos.

Método burbuja

```

public void Bubble() {
/*
 * Método burbuja, ordena los elementos de forma lineal
 *
 * Consiste en intercambiar de lugar elementos que estén inmediatamente al lado, según el criterio de
 * cual es mayor o menor. Esto se hace recorriendo todo el arreglo.
 * Esto se hace la cantidad de veces igual a la longitud del arreglo. Esto tomando en cuenta el peor
 * de los casos, por si el menor esta al final del arreglo, le tomará longitud de pasos llegar al inicio
 */
//Se hace menos uno, pues se utiliza un j+1 y no queremos provocar un error de índices
for(int i = 0; i < ArregloBubble.length -1; i++) {
    for(int j = 0; j < ArregloBubble.length - 1; j++) {
        if(ArregloBubble[j] > ArregloBubble[j+1]) {
            int aux = ArregloBubble[j];
            ArregloBubble[j] = ArregloBubble[j+1];
            ArregloBubble[j+1] = aux;
        }
    }
}
}

```

QuickSort

```
public void QuickSort(int[] a, int izq, int der) {
    /*
     * El metodo recibe un arreglo y un rango, en base a los dos punteros/numeros que se reciben
     * El contador se incrementa para saber cuantas veces se lalma al metodo
     * Asigno el pivot al primer elemento segun el rango elegido
     * Asigno los valores recibidos a variables locales, que estaremos modificando en lo largo
     * del proceso
     */

    int pivote = a[izq];
    int left = izq;
    int right = der;
    int aux;

    //Mientras el limite izquierdo sea menor al limite derecho, se ejecutara este ciclo
    //Cuando es igual, significa que ya se compararon todos los numeros del arreglo
    while (left < right) {
        //Mientras el numero evaluado sea menor o igual que el pivote, el limite izquierdo avanza
        while (a[left] <= pivote && left < right) {
            left++;
        }
        //Mientras el numero evaluado sea mayor que el pivote, el limite derecho disminuye
        while (a[right] > pivote) {
            right--;
        }

        //Si el limite izquierdo es menor al limite derecho entonces se intercambian los valores
        //de los arreglos en estas posiciones
        if (left < right) {
            aux = a[left];
            a[left] = a[right];
            a[right] = aux;
        }

        //Seguimos comparando si left < right por que esto puede cambiar dentro del while
    }

    //Luego intercambiamos los extremos del arreglo
    a[izq] = a[right];
    a[right] = pivote;

    //esto define el rango de la siguiente iteracion recursiva
    if (izq < right - 1) {
        //Si el limite inicial izquierdo es menor a el limite derecho actual menos 1, entonces se vuelve a ciclar
        //Esto permite ciclar solo si hay datos que ordenar
        QuickSort(a, izq, right - 1);
    }
    if (right + 1 < der) {
        //Si el limite inicial derecho es mayor a el limite derecho actual mas 1, entonces se vuelve a ciclar
        //Esto permite ciclar solo si hay datos que ordenar
        QuickSort(a, right + 1, der);
    }
}
```



ShellSort

```
public void ShellSort(int[] a) {  
  
    //Variables locales, longitud del arreglo, contador, un auxiliar y una bandera  
    int x, i, aux;  
    boolean b;  
    x = a.length;  
  
    //Mientras la longitud del arreglo sea mayor a cero, o sea, mientras el arreglo tenga al menos dos datos  
    //que ordenar  
    while(x > 0){  
        // se divide el arreglo en dos, para poder comparar  
        x = x / 2;  
        b = true; //Se asigna valor true a la bandera para poder continuar en el siguiente ciclo  
  
        while(b){  
            //Se vuelve false la bandera, para que no cicle infinitamente, sino bajo ciertas condiciones  
            b = false;  
            i = 0; //reiniciamos el contador  
  
            while ((i+x) <=a.length-1){ //Se asegura que nuestro avance en los indices sea menor a la longitud  
                //El ultimo es nuestro pivote  
                if (a[i] > a[i + x]){ //Se compara la primera mitad del arreglo con la segunda mitad  
                    //x marca la division del arreglo entre la mitad derecha e izquierda  
                    // Luego se hace un intercambio, para garantizar que el menor este a la mitad izquierda  
                    aux = a[i];  
                    a[i] = a[i+x];  
                    a[i+x] = aux;  
                    b = true; //Se vuelve true la bandera para que vuelva a ciclarse  
                }  
                i = i +1;  
                //avanzamos en el arreglo, nuestro contador  
            } //Cuando deja de ciclarse manda el control al ciclo anterior  
/*  
 * Si se vuelve a ejecutar entonces reinicia el contador y hace otra pasada para asegurarse  
 * Si no se vuelve a ejecutar, entonces cede el control al siguiente ciclo  
 */  
        }  
        /*  
         * Si se vuelve a ejecutar, entonces se reduce el tamano de x, por tanto se hacen cada vez  
         * Mas comparaciones, y cada vez mas cercanas, hasta comparar 1 a 1  
 */  
    }  
}
```



MergeSort

```

public void MergeSort() {
    //Primero declaramos nuestro contador, por asi llamarlo, el cual se aumentara segun la condicion que presentamos anteriormente
    int k = 1;
    do { //Este es el ciclo que permitira ciclarnos hasta que la condicion cumpla
        //Se inicializan los arreglos en cada ciclo para resetearlos, junto con las demás variables
        //Los arreglos son dinamicos, pues el calcular su tamaño en cada iteracion es tiempo muerto

        izquierda = new ArrayList<>();
        derecha = new ArrayList<>();
        int a = 0;
        int b = 0;
        int t = 0;int w = 0;
        int y = 0;
        int id = 0;

        //Este primer while permite separar el arreglo original en dos
        while(t < longitud) {
            //w y y establecen un rango en el arreglo original que se copiara en los arreglos auxiliares
            w = y;
            y = y + k;
            for(int j = w; j < y; j++) {
                /*
                 * Este for guarda con una condicion dichos elementos en los arreglos
                 */
                if(t < longitud) {
                    if(id == 0) {
                        izquierda.add(ArregloMerge[j]);
                    } else {
                        derecha.add(ArregloMerge[j]);
                    }
                    t++;
                } else {
                    break;
                }
            }
            //Este cambio de valor con condiciones permite cambiar entre los arreglos auxiliares
            if(id == 0) {
                id = 1;
            } else {
                id = 0;
            }
        }

        //Aqui es donde ordenamos
        int cont = 0; //Este contador ayuda a no pasarnos de la longitud del arreglo
        while(a < izquierda.size() && b < derecha.size() && cont < longitud - 1) {

            /*
             * Ahora analizamos cual de los elementos en los arreglos esta el menor, entonces al encontrarlo
             * lo guarda en el arreglo original
             */
            if(izquierda.get(a) <= derecha.get(b)) {
                ArregloMerge[cont] = izquierda.get(a);
                a++;
            } else {
                ArregloMerge[cont] = derecha.get(b);
                b++;
            }
            cont++;

            /*
             * Estas siguientes condiciones se ejecutan cuando un arreglo auxiliar ya se copio en el
             * original ya ordenado, y como el otro ya no tiene elementos para comparar entonces se
             * copia solo lo que falta
             */
            if(a == izquierda.size() && b < derecha.size()) {
                for(int i = cont; i < longitud; i++) {
                    ArregloMerge[i] = derecha.get(b);
                    b++;
                }
            }
            if(b == derecha.size() && a < izquierda.size()) {
                for(int i = cont; i < longitud; i++) {
                    ArregloMerge[i] = izquierda.get(a);
                    a++;
                }
            }
        }

        k = k*2;
    } while(k < longitud);
}

```

```
//Método auxiliar para crear arreglos con elementos aleatorios
public int[] Random(int l) {
    int[] x = new int[l];
    for(int i = 0; i < l; i++) {
        x[i] = (int) (Math.random()*l);
    }
    return x;
}

//Método auxiliar para imprimir arreglos
public void Peek(int[] arr){
    for(int k = 0; k < arr.length; k++){
        if(k%20 == 0){
            P("");
        }
        System.out.print(arr[k] + " ");
    }
    P("");
}

//Método auxiliar para leer cadenas
public String LeerS(){
    Scanner ls = new Scanner(System.in);
    String l = ls.nextLine();
    return l;
}

//Método auxiliar de impresión
public void P(String mensaje) {
    System.out.println(mensaje);
}
```

Estos son los métodos auxiliares que se usaron.

El primero es el que crea el arreglo, recibe como parámetro la longitud del arreglo y devuelve un arreglo de esa longitud.

El segundo es para imprimir un arreglo, le pasas como parámetro un arreglo.

El tercero es un método auxiliar para leer los enter, y el último es un método auxiliar de impresion.



Ahora presentaré cómo funciona el programa.

Se hará la comparación con 30 arreglos.
 Cada arreglo tendrá 10000 elementos aleatorios.
 Cada arreglo se presenta al inicio de cada ciclo.
 Arreglo a ordenar.

Primero se imprimen explicaciones e instrucciones al usuario sobre el funcionamiento del programa.

```

389 24 996 357 658 690 919 363 920 377 286 635 951 71 359 682 887 503 390 861
987 235 774 896 944 195 945 383 90 721 476 10 828 786 980 48 93 384 386 706
454 59 100 83 704 140 356 808 453 540 800 919 263 928 972 168 346 639 898 750
689 966 427 386 769 359 323 817 657 634 345 993 434 499 749 714 312 567 663 998
736 592 815 218 78 791 930 942 206 14 913 412 352 981 646 858 211 972 786 140
132 176 495 773 594 342 424 32 900 534 588 653 172 544 3 656 278 953 85 932
269 935 837 527 295 643 802 27 710 398 604 30 278 991 665 473 953 113 317 493
706 281 139 573 284 697 785 959 541 116 969 725 249 179 554 404 863 627 878 794
754 513 565 432 856 781 55 195 497 593 10 717 425 393 570 570 679 960 142 701
29 662 67 206 141 689 2 336 497 315 345 596 383 124 328 867 906 690 708 774
119 451 112 943 795 339 172 328 298 299 561 574 295 993 916 231 777 830 433 687
946 959 577 896 63 25 5 326 280 280 249 74 796 123 145 742 443 255 318 523
911 421 233 908 433 136 472 561 642 892 925 222 735 89 519 922 806 324 984 421
416 754 862 542 32 943 685 968 971 984 758 184 865 724 510 441 482 588 708 30
926 847 147 689 880 178 611 656 542 5 2 774 367 430 525 10 534 521 746 972
301 356 161 307 81 484 599 342 81 417 473 981 840 352 370 654 855 552 951 77
743 747 474 668 84 462 485 746 153 375 422 56 263 731 910 722 300 298 514 481
369 609 600 933 171 4 730 584 808 548 719 643 295 992 65 936 935 817 999 757
67 810 772 185 439 779 148 204 580 781 277 945 144 666 260 518 95 848 743 179
24 930 842 37 616 259 517 116 492 496 307 340 214 45 956 941 248 521 67 839
46 502 370 815 258 459 572 649 234 126 63 956 645 547 914 152 35 716 722 836
556 12 866 883 977 599 626 733 293 612 457 638 884 181 428 182 278 321 454 951
966 288 499 81 632 245 369 334 408 996 53 467 434 106 836 918 121 916 563 254
949 880 497 402 588 130 200 421 386 398 736 439 986 243 275 452 984 27 892 866
992 263 929 17 471 730 803 422 972 247 563 191 203 108 799 835 461 943 740 707
531 249 510 440 115 866 544 33 200 468 97 714 569 529 652 273 824 261 768 369
860 779 578 181 748 73 704 907 355 443 860 790 837 693 742 775 359 750 151 437
998 371 437 509 815 223 472 201 492 984 375 531 794 14 98 21 320 834 825 474
676 872 832 12 250 907 123 182 217 216 645 758 283 680 381 792 925 10 920 589
598 807 391 533 406 402 796 122 290 263 838 134 625 330 338 580 446 776 181 994
246 561 541 653 354 644 827 501 482 853 378 942 909 242 443 149 127 217 408 224
509 835 531 349 698 476 668 85 984 493 634 68 864 570 834 680 380 223 775 711
582 335 906 630 624 623 991 860 629 691 879 344 390 663 94 803 434 459 170 275
707 404 997 958 413 265 762 875 111 754 242 620 204 226 317 282 395 584 569 650
502 389 742 996 259 785 665 20 409 122 954 71 396 109 439 343 470 914 817 587
841 860 434 794 832 298 503 430 609 777 258 63 196 844 763 638 937 698 125 229
483 435 838 841 675 810 889 704 402 283 7 564 631 609 166 652 331 395 10 409
915 855 423 501 202 433 825 387 587 145 137 473 303 193 551 919 163 911 392 253
180 144 375 280 769 929 526 206 62 748 862 496 2 684 692 778 484 510 65 974
802 154 445 981 41 956 182 379 878 863 717 106 437 25 752 526 486 267 799 51
260 545 637 313 628 825 190 918 170 781 266 955 946 320 417 775 400 243 879 49
888 267 212 712 776 221 777 235 928 501 634 604 655 745 793 465 900 257 14 214
480 941 488 162 745 342 836 539 854 973 802 788 217 221 911 528 332 44 158 571
362 100 461 593 342 148 241 441 799 346 973 522 915 546 962 422 128 813 949 729
148 156 322 0 0 917 891 950 580 330 182 256 651 536 163 796 904 714 173 758
959 72 564 817 691 151 225 503 318 250 754 944 47 840 469 412 879 491 877 483
396 194 482 161 963 945 994 34 785 124 134 356 699 503 204 995 419 682 508 332
174 384 553 533 331 297 636 89 665 123 99 173 365 34 46 131 744 38 747 401
911 604 351 241 296 516 713 921 48 73 372 447 777 697 736 463 39 473 526 706
237 382 400 284 341 787 481 75 925 261 175 765 562 991 305 388 517 774 45 737

```

A continuación imprime el arreglo para que el usuario pueda verlo y comprobar que es uno distinto en cada prueba

```
Burbuja Ordenado...
BubbleSort: 16
Quicksort Ordenado...
QuickSort: 0
ShellSort Ordenado...
ShellSort: 0
MergeSort Ordenado...
MergeSort: 15
Presione enter para avanzar a la siguiente prueba...
```

Cada vez que un método termina de ejecutarse se imprime que ha terminado, luego imprime el tiempo en milisegundos.

Ahora un detalle. El número 0 no es real, es una representación de la computadora para decir que es muy pequeño. El tiempo originalmente estaba en nanosegundos, y así se apreciaba bien, hasta que hacías arreglos grandes y te tenia que representar el método burbuja como un número*e.

Se debe presionar enter para poder avanzar a la siguiente prueba, así puedes analizar el arreglo y los tiempos sin que estos avancen rápido.

```
Promedio de los tiempos de ejecucion.
QuickSort: 0.0
MergeSort: 0.0
ShellSort: 1.0
BubbleSort: 13.0
Presione enter para salir.
```

Al final de todas las ejecuciones realiza un promedio y despliega los tiempos. Aumentando el número de elementos en los arreglos notamos cómo cambian los tiempos.

```
Promedio de los tiempos de ejecucion.
ShellSort: 0.0
MergeSort: 2.0
QuickSort: 16.0
BubbleSort: 51.0
Presione enter para salir.
```

Como vemos, los tiempos cambian, y las posiciones en rapidez también.

Conclusión:

Después de muchas veces realizar las 30 pruebas concluyo que el método ShellSort es el método de ordenamiento más rápido de estos cuatro presentados, en cuanto a cantidades grandes de números. Sin embargo, en ocasiones debido al acomodo de los números en los arreglos y el tamaño de estos, el método más rápido cambia, y es por que cada uno tiene una característica que lo hace más veloz en algunos momentos. Y, más que nada, es la práctica y utilización de estos la que nos permitirá determinar cuál necesitamos, además de otras condiciones.