



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Edgar Tista García

Asignatura: Estructura de datos y algoritmos II -1317

Grupo: 10

No. de práctica(s): 1

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 25

Semestre: 2023-1

Fecha de entrega: 28 de agosto del 2022

Observaciones:

CALIFICACIÓN: _____



Práctica 1.

Algoritmos de ordenamiento. Parte 1.

OBJETIVOS

- **Objetivo general:** El estudiante identificará la estructura de los algoritmos de ordenamiento InsertionSort, SelectionSort y BubbleSort.
- **Objetivo de la clase:** El alumno revisará el uso de bibliotecas como capas de abstracción, así como realización de pruebas de los algoritmos para diferentes instancias de colecciones para conocer la complejidad de estos.

DESARROLLO

Ejercicio 1. Análisis inicial

El concepto de capas de abstracción permite al programador elaborar sus propias bibliotecas basado en la solución de problemas específicos para poder elaborar soluciones más complejas.

- a) Verifica los archivos proporcionados, junto con sus respectivas implementaciones:

✓ **utilidades.h**

Se encuentra la definición de las funciones *swap()*, *printArray()* y *printSubArray()*, las cuales serán de utilidad para la construcción de los algoritmos de ordenamiento y su visualización.

✓ **utilidades.c**

Contiene las implementaciones de las funciones definidas en *utilidades.h*.



swap():

```
void swap(int *a, int *b) {  
    int t = *a;  
    *a = *b;  
    *b = t;  
}
```

Imagen 1.1. Función swap()

Descripción breve de la función: Intercambia el contenido de dos referencias en memoria con datos tipo *int*.

Parámetro “a”: Primer elemento que se desea intercambiar.

Parámetro “b”: Segundo elemento que se desea intercambiar.

printArray():

```
void printArray(int arr[], int size) {  
    int i;  
    for (i = 0; i < size; i++)  
        printf("%d ", arr[i]);  
    printf("\n");  
}
```

Imagen 1.2. Función printArray()

Descripción breve de la función: Imprime un arreglo de enteros.

Parámetro “arr”: Arreglo de datos tipo *int*.

Parámetro “size”: Cantidad de elementos del arreglo.



printSubArray():

```
void printSubArray(int arr[], int low, int high) {  
    int i;  
    printf("Sub array : ");  
    for (i = low; i ≤ high; i++)  
        printf("%d ", arr[i]);  
    printf("\n");  
}
```

Imagen 1.3. Función printSubArray()

Descripción breve de la función: Imprime un subarreglo de un arreglo de enteros.

Parámetro “low”: Índice inicial del subarreglo.

Parámetro “high”: Índice final del subarreglo.

✓ ordenamientos.h

Se encuentra la definición de las funciones *selectionSort()* e *insertionSort()*, ambos, los algoritmos de ordenamiento vistos en clase.

✓ ordenamientos.c

Contiene las implementaciones de los algoritmos definidos en *ordenamientos.h*. Como aspecto a destacar, la diferencia entre los pseudocódigos vistos en clase y la implementación en C de estos es la modificación realizada para incluir el índice cero de un arreglo dado.



selectionSort():

```
void selectionSort(int arreglo[], int n) {  
    int indiceMenor, i, j;  
    for (i = 0; i < n - 1; i++) {  
        indiceMenor = i;  
        for (j = i + 1; j < n; j++) {  
            if (arreglo[j] < arreglo[indiceMenor])  
                indiceMenor = j;  
        }  
        if (i != indiceMenor) {  
            swap(&arreglo[i], &arreglo[indiceMenor]);  
        }  
        printf("\nIteracion numero %d \n", i + 1);  
        printArray(arreglo, n);  
    }  
}
```

Imagen 1.4. *selectionSort()*

Descripción del algoritmo: Consiste en revisar todos los elementos y seleccionar el valor más pequeño de la colección, para ello, realiza muchas comparaciones empezando por el elemento de la extrema izquierda: recorre todos los elementos siguientes hasta encontrar el menor. Luego, se apoya en la variable *indiceMenor* para intercambiarlo por el primero. Continúa con el siguiente índice y se repite para los demás $n-1$ elementos del arreglo.

Parámetro “arreglo”: Arreglo de enteros.

Parámetro “n”: Cantidad de elementos del arreglo.



insertionSort():

```
void insertionSort(int a[], int n) {  
    int i, j, k;  
    int aux;  
    for (i = 1; i < n; i++) {  
        j = i;  
        aux = a[i];  
        while (j > 0 && aux < a[j - 1]) {  
            a[j] = a[j - 1];  
            j--;  
        }  
        a[j] = aux;  
        printf("\nIteracion numero %d \n", i);  
        printArray(a, n);  
    }  
}
```

Imagen 1.5. *insertionSort()*

Descripción del algoritmo: Revisa uno por uno los elementos del arreglo. Conforme avanza en la colección, cada dato se compara con los elementos a su izquierda, para ello se parte del supuesto de que los elementos a la izquierda ya están acomodados, luego, busca su posición en el subarreglo de la izquierda y lo inserta.

Parámetro “a”: Arreglo de enteros.

Parámetro “n”: Cantidad de elementos del arreglo.

b) Agrega en los archivos correspondientes el código del algoritmo BubbleSort

```
void bubbleSort(int a[], int size){  
    int i,j,n;  
    n= size;  
    for(i=n-1;i>0;i--){  
        for(j=0; j<i; j++){  
            if(a[j]>a[j+1])  
                swap(&a[j], &a[j+1]);  
        }  
    }  
}
```



- c) Modifica el código para que bubble sort se detenga en el momento que la lista se encuentre ordenada.

```
void bubbleSort(int a[], int size) {  
    int i, j, n, exchange;  
    n = size;  
    for (i = n - 1; i > 0; i--) {  
        exchange = 0;  
        for (j = 0; j < i; j++) {  
            if (a[j] > a[j + 1]) {  
                swap(&a[j], &a[j + 1]);  
                exchange = 1;  
            }  
        }  
        if (exchange == 0) {  
            break;  
        }  
    }  
}
```

Imagen 1.6. Modificación de bubbleSort()

Ejercicio 2. Probando los ordenamientos

- a) Crea un nuevo proyecto para lenguaje c (Puede ser un proyecto de Xcode, codeblocks, o Dev c++). Elabora un programa en el que, utilizando las bibliotecas proporcionadas, utilices los algoritmos que se encuentran en la biblioteca de ordenamientos.

- ✓ Se deberá crear un arreglo de 20 elementos, para ello se dará al usuario la opción de ingresar los valores o bien, que se llenen con valores aleatorios (0-999).
- ✓ Mediante un menú de usuario, se podrá elegir El algoritmo de ordenamiento deseado. (insertion, selection, bubble)
- ✓ Una vez elegido, el programa deberá mostrar las modificaciones que se hacen al arreglo en cada iteración y el arreglo final ordenado.



b) Realiza la verificación con su respectiva evidencia de que la modificación realizada a bubble sort funciona correctamente.

✓ Selecciona la opción de ingresar valores en el arreglo y coloca valores que ya se encuentren ordenados, al ejecutar el programa, debería realizar una sola pasada, ya que al estar ordenado, no debe seguir realizando el resto de las iteraciones.

Capturas de pantalla de la ejecución del programa

```
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P1\Ejercicio 2> gcc utilidades.c ordenamientos.c main.c -o main.exe
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P1\Ejercicio 2> ./main
-----Selecciona una opcion-----
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
1

-----Selecciona el tipo de llenado del arreglo-----
1. Numeros aleatorios (0-99)
2. Ingresar numeros manualmente
█
```

Imagen 2.1.1. Compilación y menú del programa



The image shows a Visual Studio Code interface with a terminal window open. The terminal displays the output of a C program. The program starts by asking the user to select a type of array filling: "1. Numeros aleatorios (0-99)" or "2. Ingresar numeros manualmente". The user has entered "1". The program then uses Insertion Sort to sort the array. It displays the array after each of 7 iterations. The background of the terminal window features a large, stylized anime-style illustration of a girl with blue hair and a red flower. The Visual Studio Code interface includes a sidebar on the left with icons for Explorer, Search, Source Control, Run and Debug, and Extensions. The top of the window shows the menu bar (File, Edit, Selection, View, Go, Run, Terminal, Help) and the title bar (main.c - Programacion - Visual Studio Code). The bottom status bar shows the current line and column (Ln 28, Col 21), the number of spaces (4), the encoding (UTF-8), the line ending (CRLF), the file type (C), the window state (Win32), and the user's name (Rikka).



The image shows a Visual Studio Code interface with a terminal window open. The terminal displays a loop of 17 iterations, each printing a sequence of 25 numbers. The numbers in each sequence are: 282, 283, 391, 560, 838, 914, 918, 933, 861, 600, 117, 889, 393, 535, 25, 644, 945, 661, 587, followed by a blank space and then 70. The iterations are labeled from 8 to 17. The background of the terminal window features a large, stylized illustration of an anime character with blue hair and a red flower. The Visual Studio Code interface includes a menu bar at the top with options like File, Edit, View, Go, Run, Terminal, and Help. The terminal window title is 'main.c - Programacion - Visual Studio Code'. The status bar at the bottom shows 'Ln 28, Col 21', 'Spaces: 4', 'UTF-8', 'CRLF', 'C', 'Win32', and 'Rikka'.



ESTRUCTURA DE DATOS Y ALGORITMOS II



```
File Edit Selection View Go Run Terminal Help main.c - Programacion - Visual Studio Code
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
70 117 282 283 391 560 600 838 861 914 918 933 889 393 535 25 644 945 661 587
Iteracion numero 12
70 117 282 283 391 560 600 838 861 889 914 918 933 393 535 25 644 945 661 587
Iteracion numero 13
70 117 282 283 391 393 560 600 838 861 889 914 918 933 535 25 644 945 661 587
Iteracion numero 14
70 117 282 283 391 393 535 560 600 838 861 889 914 918 933 25 644 945 661 587
Iteracion numero 15
25 70 117 282 283 391 393 535 560 600 838 861 889 914 918 933 644 945 661 587
Iteracion numero 16
25 70 117 282 283 391 393 535 560 600 644 838 861 889 914 918 933 945 661 587
Iteracion numero 17
25 70 117 282 283 391 393 535 560 600 644 838 861 889 914 918 933 945 661 587
Iteracion numero 18
25 70 117 282 283 391 393 535 560 600 644 661 838 861 889 914 918 933 945 587
Iteracion numero 19
25 70 117 282 283 391 393 535 560 587 600 644 661 838 861 889 914 918 933 945
El arreglo final ordenado es:
25 70 117 282 283 391 393 535 560 587 600 644 661 838 861 889 914 918 933 945
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P1\Ejercicio 2>
```

Imagen 2.1.4. Arreglo final usando Insertion Sort

```
File Edit Selection View Go Run Terminal Help main.c - Programacion - Visual Studio Code
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
-----Selecciona una opcion-----
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
2
-----Selecciona el tipo de llenado del arreglo-----
1. Numeros aleatorios (0-99)
2. Ingresar numeros manualmente
1
-----Utilizando Selection Sort-----
El arreglo es:
928 78 56 417 7 694 172 228 19 492 753 17 513 114 801 91 33 631 818 453
Iteracion numero 1
7 78 56 417 928 694 172 228 19 492 753 17 513 114 801 91 33 631 818 453
Iteracion numero 2
7 17 56 417 928 694 172 228 19 492 753 78 513 114 801 91 33 631 818 453
Iteracion numero 3
7 17 19 417 928 694 172 228 56 492 753 78 513 114 801 91 33 631 818 453
Iteracion numero 4
7 17 19 33 928 694 172 228 56 492 753 78 513 114 801 91 417 631 818 453
Iteracion numero 5
7 17 19 33 56 694 172 228 928 492 753 78 513 114 801 91 417 631 818 453
```

Imagen 2.2.1. Usando Selection Sort



ESTRUCTURA DE DATOS Y ALGORITMOS II

```
File Edit Selection View Go Run Terminal Help main.c - Programacion - Visual Studio Code
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
Iteracion numero 6
7 17 19 33 56 78 172 228 928 492 753 694 513 114 801 91 417 631 818 453
Iteracion numero 7
7 17 19 33 56 78 91 228 928 492 753 694 513 114 801 172 417 631 818 453
Iteracion numero 8
7 17 19 33 56 78 91 114 928 492 753 694 513 228 801 172 417 631 818 453
Iteracion numero 9
7 17 19 33 56 78 91 114 172 492 753 694 513 228 801 928 417 631 818 453
Iteracion numero 10
7 17 19 33 56 78 91 114 172 228 753 694 513 492 801 928 417 631 818 453
Iteracion numero 11
7 17 19 33 56 78 91 114 172 228 417 694 513 492 801 928 753 631 818 453
Iteracion numero 12
7 17 19 33 56 78 91 114 172 228 417 453 513 492 801 928 753 631 818 694
Iteracion numero 13
7 17 19 33 56 78 91 114 172 228 417 453 492 513 801 928 753 631 818 694
Iteracion numero 14
7 17 19 33 56 78 91 114 172 228 417 453 492 513 801 928 753 631 818 694
Iteracion numero 15
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 928 753 801 818 694
Ln 28, Col 21 Spaces: 4 UTF-8 CRLF C Win32 Rikka
```

Imagen 2.2.2. Iteraciones de Selection Sort

```
File Edit Selection View Go Run Terminal Help main.c - Programacion - Visual Studio Code
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
7 17 19 33 56 78 91 114 172 228 417 694 513 492 801 928 753 631 818 453
Iteracion numero 12
7 17 19 33 56 78 91 114 172 228 417 453 513 492 801 928 753 631 818 694
Iteracion numero 13
7 17 19 33 56 78 91 114 172 228 417 453 492 513 801 928 753 631 818 694
Iteracion numero 14
7 17 19 33 56 78 91 114 172 228 417 453 492 513 801 928 753 631 818 694
Iteracion numero 15
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 928 753 801 818 694
Iteracion numero 16
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 694 753 801 818 928
Iteracion numero 17
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 694 753 801 818 928
Iteracion numero 18
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 694 753 801 818 928
Iteracion numero 19
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 694 753 801 818 928
El arreglo final ordenado es:
7 17 19 33 56 78 91 114 172 228 417 453 492 513 631 694 753 801 818 928
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P1\Ejercicio 2>
Ln 28, Col 21 Spaces: 4 UTF-8 CRLF C Win32 Rikka
```

Imagen 2.2.3. Arreglo final usando Selection Sort



ESTRUCTURA DE DATOS Y ALGORITMOS II

Capturas de pantalla de la ejecución de Bubble Sort

```
main.c - Programacion - Visual Studio Code
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
-----Selecciona una opcion-----
1. Insertion Sort
2. Selection Sort
3. Bubble Sort
3

-----Selecciona el tipo de llenado del arreglo-----
1. Numeros aleatorios (0-99)
2. Ingresar numeros manualmente
2
Ingresa el numero 1: 1
Ingresa el numero 2: 2
Ingresa el numero 3: 3
Ingresa el numero 4: 4
Ingresa el numero 5: 5
Ingresa el numero 6: 6
Ingresa el numero 7: 7
Ingresa el numero 8: 8
Ingresa el numero 9: 9
Ingresa el numero 10: 10
Ingresa el numero 11: 11
Ingresa el numero 12: 12
Ingresa el numero 13: 13
Ingresa el numero 14: 14
Ingresa el numero 15: 15
Ingresa el numero 16: 16
Ingresa el numero 17: 17
Ingresa el numero 18: 18
Ingresa el numero 19: 19
```

Imagen 2.3.1. Ingresando números ordenados manualmente

```
main.c - Programacion - Visual Studio Code
PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE
2
Ingresa el numero 1: 1
Ingresa el numero 2: 2
Ingresa el numero 3: 3
Ingresa el numero 4: 4
Ingresa el numero 5: 5
Ingresa el numero 6: 6
Ingresa el numero 7: 7
Ingresa el numero 8: 8
Ingresa el numero 9: 9
Ingresa el numero 10: 10
Ingresa el numero 11: 11
Ingresa el numero 12: 12
Ingresa el numero 13: 13
Ingresa el numero 14: 14
Ingresa el numero 15: 15
Ingresa el numero 16: 16
Ingresa el numero 17: 17
Ingresa el numero 18: 18
Ingresa el numero 19: 19
Ingresa el numero 20: 20

-----Utilizando Bubble Sort-----
El arreglo es:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

El arreglo final ordenado es:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P1\Ejercicio 2>
```

Imagen 2.3.2. Arreglo final usando Bubble Sort



Observaciones: Con el arreglo ordenado en su estado inicial, el algoritmo no realizó ninguna iteración, por lo tanto, la implementación es correcta.

Ejercicio 3. Análisis de la complejidad computacional

- ✓ Agrega en los algoritmos de ordenamiento, las instrucciones necesarias para contabilizar el número de operaciones del algoritmo (comparaciones, intercambios, inserciones) que sean considerables en el análisis de complejidad temporal.
- ✓ Realiza pruebas con arreglos de diferente tamaño (llenar los valores con números aleatorios).
 - 50, 100, 500, 800, 1000, 2000*, 5000*, 10,000*
- ✓ Realiza al menos 5 ejecuciones en cada uno de los tamaños mencionados y muestra una tabla general con promedio del número de operaciones en cada uno.
- ✓ Realiza gráficas de la cantidad de operaciones que se realizan de acuerdo con el tamaño de la entrada y explica tus resultados.
- ✓ En estas gráficas observarás la complejidad de cada algoritmo.
- ✓ Deberás agregar capturas de pantalla de los aspectos más relevantes de las pruebas realizadas.



Instrucciones agregadas a los algoritmos

NOTA: Se comentaron las líneas que imprimen las iteraciones para fines prácticos.

```
void selectionSort(int arreglo[], int n) {  
    int indiceMenor, i, j;  
    unsigned long long int ops = 0;  
    for (i = 0; i < n - 1; i++) {  
        indiceMenor = i;  
        for (j = i + 1; j < n; j++) {  
            ops++;  
            if (arreglo[j] < arreglo[indiceMenor])  
                indiceMenor = j;  
        }  
        ops++;  
        if (i != indiceMenor) {  
            ops++;  
            swap(&arreglo[i], &arreglo[indiceMenor]);  
        }  
        // printf("\nIteracion numero %d \n", i + 1);  
        // printArray(arreglo, n);  
    }  
    printf("\nNo. de operaciones: %llu\n", ops);  
}
```

Imagen 3.1.1. Modificación a selectionSort()

- Variable tipo *unsigned long long int* para contabilizar las operaciones
- Comparación
- Comparación
- Intercambio
- Imprime la cantidad de operaciones obtenidas al final del ordenamiento



```
void insertionSort(int a[], int n) {  
    int i, j, k;  
    int aux;  
    unsigned long long int ops = 0;  
    for (i = 1; i < n; i++) {  
        j = i;  
        aux = a[i];  
        while (j > 0 && aux < a[j - 1]) {  
            ops++;  
            a[j] = a[j - 1];  
            ops++;  
            j--;  
        }  
        ops++;  
        a[j] = aux;  
        // printf("\nIteracion numero %d \n", i);  
        // printArray(a, n);  
    }  
    printf("\nNo. de operaciones: %llu\n", ops);  
}
```

Imagen 3.1.2. Modificación a insertionSort()

Imprime la cantidad de operaciones obtenidas al final del ordenamiento

Comparación

Intercambio

Inserción

Variable tipo *unsigned long long int* para contabilizar las operaciones



```
void bubbleSort(int a[], int size) {  
    int i, j, n, exchange;  
    unsigned long long int ops = 0;  
    n = size;  
    for (i = n - 1; i > 0; i--) {  
        exchange = 0;  
        for (j = 0; j < i; j++) {  
            ops++;  
            if (a[j] > a[j + 1]) {  
                ops++;  
                swap(&a[j], &a[j + 1]);  
                exchange = 1;  
            }  
        }  
        ops++;  
        if (exchange == 0) {  
            break;  
        }  
        // printf("\nIteracion numero %d \n", i + 1);  
        // printArray(a, n);  
    }  
    printf("\nNo. de operaciones: %llu\n", ops);  
}
```

Imagen 3.1.3. Modificación a bubbleSort()

Variable tipo *unsigned long long int* para contabilizar las operaciones

Comparación

Intercambio

Comparación

Variable tipo *unsigned long long int* para contabilizar las operaciones

**Tablas con los valores obtenidos en las ejecuciones****Insertion Sort**

#Prueba	1	2	3	4	5	Prom.
50	1405	1349	1321	1401	1199	1335
100	4991	5703	5155	4569	4871	5058
500	123 955	122 169	123 961	137 359	125 495	126 588
800	314 195	319 607	319 077	311 191	327 309	318 276
1000	500 277	503 901	493 897	500 383	507 501	501 192
2000	1 981 221	1 985 777	1 981 097	2 023 885	1 999 115	1 994 219
5000	12 554 907	12 551 693	12 592 959	12 326 627	12 565 187	12 518 275
10000	50 354 779	49 735 755	50 678 751	49 400 155	50 388 945	50 111 677

Selection Sort

#Prueba	1	2	3	4	5	Prom.
50	1319	1320	1317	1318	1322	1319
100	5142	5147	5145	5143	5145	5144
500	125 743	125 741	125 743	125 746	125 738	125 742
800	321 189	321 190	321 190	321 192	321 193	321 191
1000	501 485	501 494	501 491	501 495	501 493	501 492
2000	2 002 991	2 002 993	2 002 993	2 002 986	2 002 988	2 002 990
5000	12 507 489	12 507 484	12 507 488	12 507 486	12 507 486	12 507 487
10000	50 014 984	50 014 978	50 014 984	50 014 983	50 014 979	50 014 982

Bubble Sort

#Prueba	1	2	3	4	5	Prom.
50	1764	1756	1807	1756	1783	1773
100	7635	7723	7437	7010	7375	7436
500	184 701	184 771	188 128	185 614	186 539	185 951
800	476 316	483 224	474 082	473 187	478 755	477 113
1000	746 955	739 296	748 062	754 096	746 156	746 913
2000	3 008 596	2 961 904	2 991 057	3 008 665	2 997 324	2 993 509
5000	18 651 938	18 663 593	18 739 634	18 672 649	18 691 957	18 683 954
10000	74 778 177	75 022 432	75 326 589	74 888 845	75 098 022	75 022 813



Gráficas de tamaño del arreglo vs operaciones

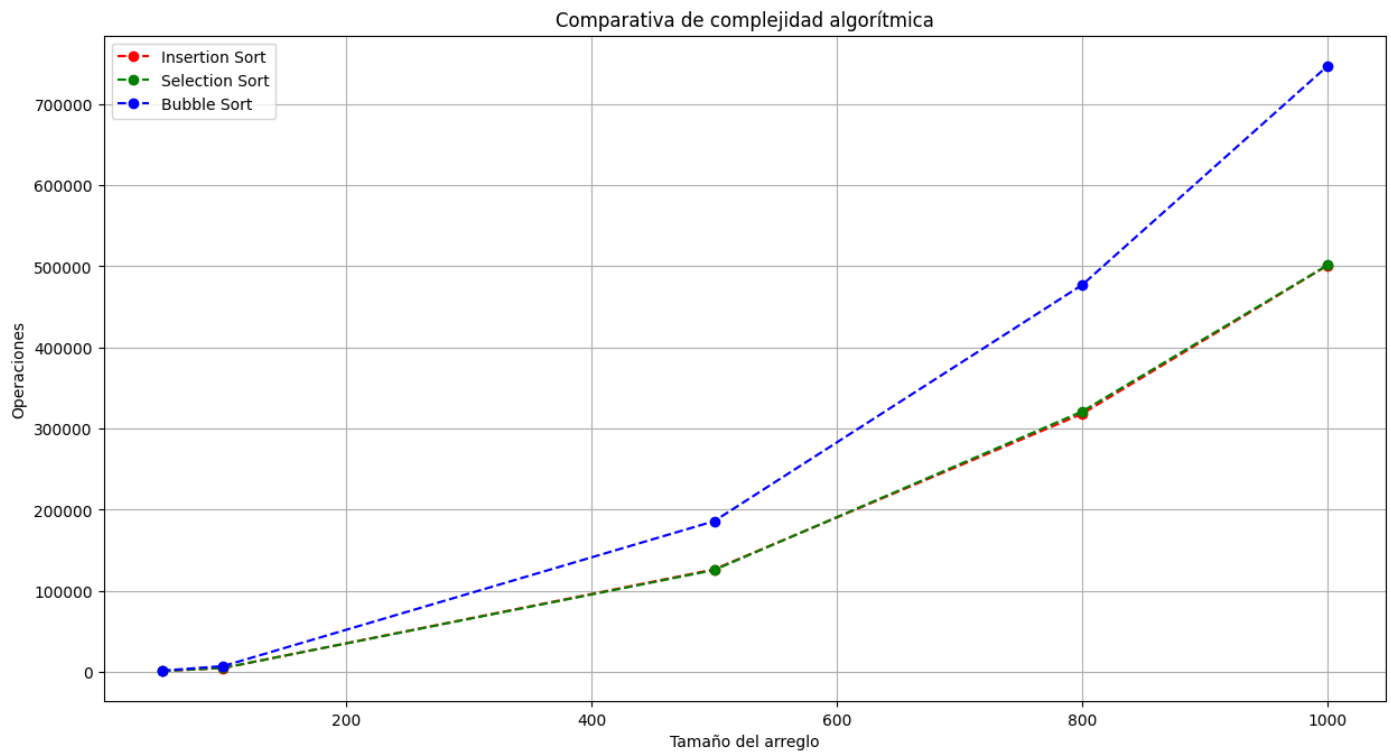


Imagen 3.2.1. Tamaños de arreglo entre 50 y 1000

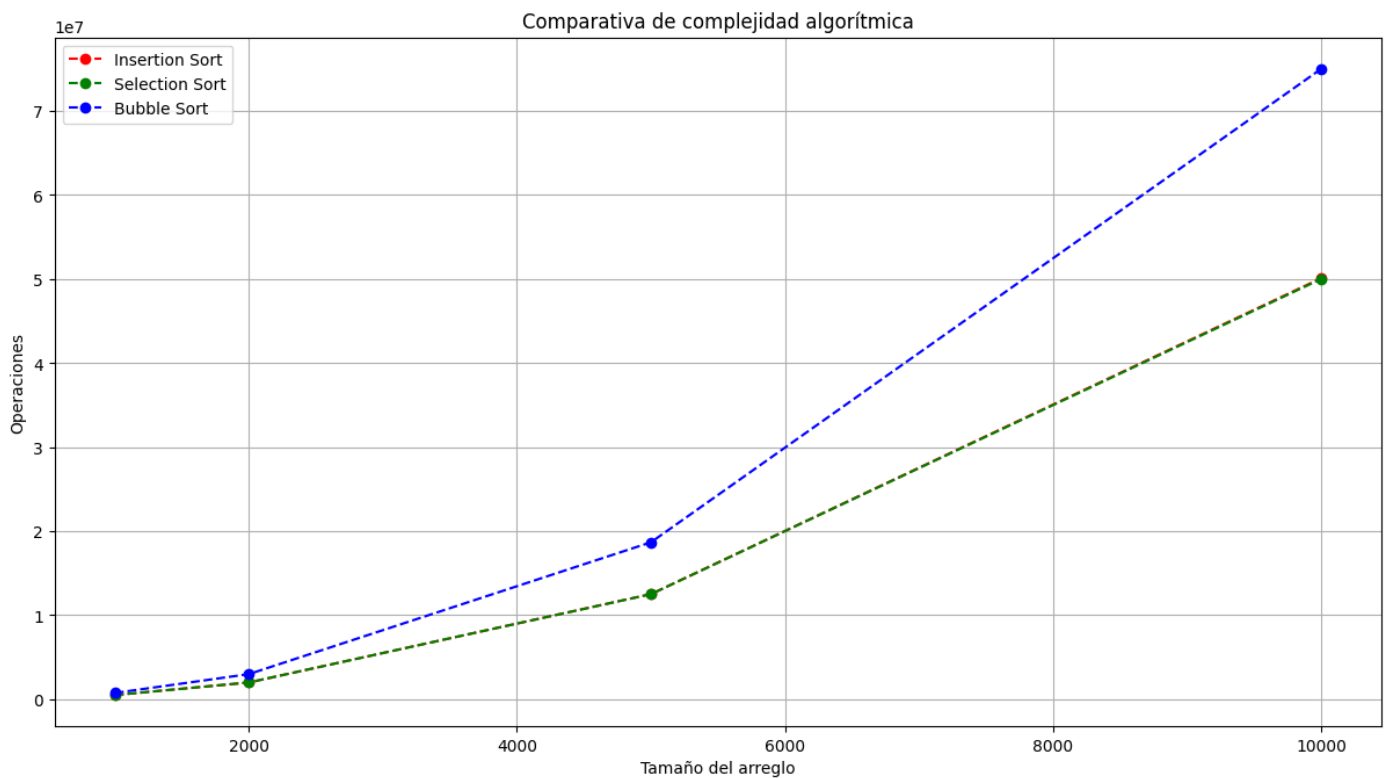


Imagen 7. Tamaños de arreglo entre 1000 y 10000



Observaciones: Insertion Sort y Selection Sort arrojaron cantidades de operaciones bastante similares. Por otra parte, Bubble Sort se aleja bastante del número de operaciones, por lo tanto, es el menos eficiente de los tres algoritmos de ordenamiento.

Es posible que la diferencia en Bubble Sort se deba a la comparación que se agregó al modificar la función. Aquí es donde encontramos dos posibilidades, la modificación de Bubble Sort reduce la complejidad de $O(n^2)$ a $O(n)$ en el mejor de los casos. Sin embargo, cuando no se da tal caso, añadimos bastantes operaciones de comparación al algoritmo.

Ejercicio 4. Ahora en Java

- ✓ Abre el proyecto “OrdenamientoP1” utilizando el editor NetBeans, **compila y ejecuta el programa** para verificar el funcionamiento.
- ✓ Revisa el código de las clases proporcionadas.
 - Explica las diferencias entre la forma en la que se manda a llamar el algoritmo de selección y el de inserción.
 - ¿Por qué en la clase principal se pueden utilizar las otras sin tener que llamarlas explícitamente?
 - Realiza los comentarios de otros aspectos que te llamen la atención o te causen dudas del código de Java (codificación, ejecución, clases, etc.).



Ejecución del proyecto

NOTA: Se utilizó el editor de texto Visual Studio Code para abrir el proyecto, y la ejecución usando las extensiones *Language Support for Java by Red Hat* y *Debugger for Java*.

```
\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDetailsInExceptionMessages'  
'-cp' 'C:\Users\cemh0\AppData\Roaming\Code\User\workspaceStorage\7b6174cf907  
524a692c5491607cdc8ca\redhat.java\jdt_ws\Ejercicio 4_97ad5d12\bin' 'Practica  
1Eda2.Principal'  
Arreglos Originales  
92 48 69 15 9 25 11 71 45 12 96 101 20 99 32 64 3 50 88 77  
29 84 96 51 90 52 11 17 54 21 80 44 2 23 46 5 66 33 99 1  
Arreglos ordenados  
3 9 11 12 15 20 25 32 45 48 50 64 69 71 77 88 92 96 99 101  
1 2 5 11 17 21 23 29 33 44 46 51 52 54 66 80 84 90 96 99  
PS D:\Users\cemh0\Programacion\3Sem\EDA II\P1\Ejercicio 4>
```

Imagen 4.1. Salida del programa



Diferencias en los llamados a los algoritmos

```
Insercion.insertionSort(arr1);  
  
Seleccion seleccion = new Seleccion();  
seleccion.selectionSort(arr2);
```

Imagen 4.2. Llamado a los algoritmos

Se accede al método `insertionSort()` de la clase `Insercion`. Se realiza el llamado al método sin instanciar la clase debido a que es un método estático (el método pertenece a la clase y no al objeto).

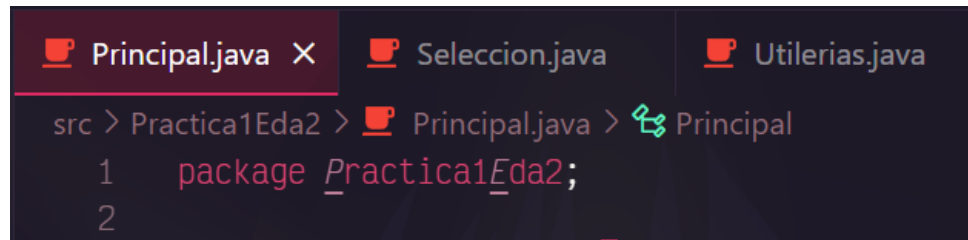
Se accede al método `selectionSort()` de la clase `Seleccion`. Para ello se crea una instancia de la clase `Seleccion` (porque el método no es estático), con la cual llamamos al método.

```
public class Insercion {  
    public static void insertionSort(int array[]) {  
        int n = array.length;  
        for (int j = 1; j < n; j++) {  
            int key = array[j];  
            int i = j - 1;  
            while ((i > -1) && (array[i] > key)) {  
                array[i + 1] = array[i];  
                i--;  
            }  
            array[i + 1] = key;  
        }  
    }  
}
```

Imagen 4.3. Insertion Sort es un método estático



Observaciones: En cada uno de los archivos del proyecto tenemos la siguiente línea en común



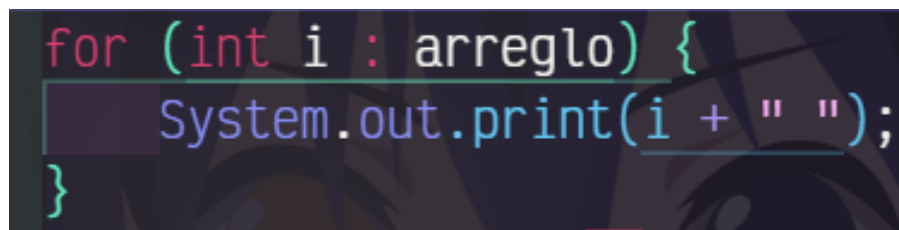
```
src > Practica1Eda2 > Principal.java > Principal
1 package Practica1Eda2;
2
```

Imagen 4.4. Paquete en Java

Esto significa que estamos agrupando todos los archivos en un mismo paquete para facilitar la modularidad del código.

La agrupación de las clases en un mismo paquete implica que todas coexisten en un mismo nivel jerárquico, y, por lo tanto, podemos llamar clases y sus respectivas funciones sin tener que llamarlas explícitamente.

Observaciones: Java soporta la estructura de repetición for-each.



```
for (int i : arreglo) {
    System.out.print(i + " ");
}
```

Imagen 4.5. Bucle for-each en Java



CONCLUSIONES

Algunos puntos que se concluyen de la primera actividad son los siguientes:

- Revisando las bibliotecas que componen al proyecto, es notable la implementación de capas de abstracción en las que se dividió el programa para resolver problemas específicos en cada una de ellas.
- Utilidades: realiza operaciones que se necesitarán en los algoritmos; ordenamientos: implementa los algoritmos de ordenamiento; y main: contiene el menú del programa.
- Se identificaron las diferencias en el funcionamiento de los tres algoritmos de ordenamiento vistos en esta práctica al explicar cada una de sus funciones.

Para el segundo ejercicio se puede concluir que:

- Los algoritmos funcionan de manera adecuada después de realizar varias pruebas usando diferentes ejemplos de arreglos.
- La modificación a Bubble Sort es funcional.

Al término del tercer ejercicio se llegó a las siguientes conclusiones:

- Insertion Sort y Selection Sort tienen una complejidad bastante similar.
- De los tres algoritmos, Bubble Sort fue el menos eficiente.
- La modificación a Bubble Sort puede mejorar la complejidad en el mejor caso, pero de lo contrario, agrega muchas operaciones al algoritmo.

Por último, los aspectos relevantes a los que se llegó con el cuarto ejercicio son:

- Se encontraron dos maneras diferentes de hacer el llamado a los algoritmos: esto se debe a la diferencia de ser un método estático en el caso de Insertion Sort, el cual no necesita crear una instancia de la clase; y Selection Sort, que no es un método estático y por lo tanto se debe crear de antemano una instancia de la clase Selecccion.
- Con el uso de paquetes en Java se pueden agrupar diferentes clases y archivos en un solo contenedor lógico, que nos permite interactuar con clases dentro del mismo paquete sin tener que llamarlas explícitamente.
- De la misma forma que en el ejercicio 1, existen diferentes capas de abstracción, que subdividen el problema y resuelven problemas más específicos en cada una de ellas.



ESTRUCTURA DE DATOS Y ALGORITMOS II



Dicho todo lo anterior, se ha cumplido con cada uno de los objetivos planteados para el desarrollo de esta práctica dado que se identificaron las diferentes estructuras de los algoritmos de ordenamiento, así como se analizó su complejidad, al mismo tiempo que se revisó el uso de bibliotecas como capas de abstracción.

Como comentario final, considero que la mayor ventaja de estos algoritmos es que son fáciles de comprender, sin embargo, no son los más eficientes