



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: TISTA GARCÍA EDGAR

Asignatura: ESTRUCTURAS DE DATOS Y ALGORITMOS II

Grupo: 5

No de Práctica(s): 02

Integrante(s): GÓMEZ LUNA ALEJANDRO

*No. de Equipo de
cómputo empleado:* 42

No. de Lista o Brigada: 42

Semestre: 2020-1

Fecha de entrega: 20/08/2019

Observaciones:

CALIFICACIÓN: _____

- **Objetivo de la práctica**
El estudiante identificará la estructura de los algoritmos de ordenamiento BubbleSort, QuickSort y HeapSort.
- **Desarrollo**
 - a) Ejercicios de la guía: Para esta práctica se analizarán los algoritmos de ordenamiento de QuickSort, HeapSort y BubbleSort.
 - a. Para QuickSort, se nos proporciona la implementación del algoritmo en Python. En esta implementación podemos observar que la función QuickSort es de tipo recursiva, ya que se llama a sí misma dos veces. Se llama a sí misma en un escenario más simple, en donde se divide la lista en otras más pequeños, es decir, hace uso de la técnica de divide y vencerás. Para saber en qué punto dividir a la lista en otras sub-listas, se hace uso de la función Particionar. La función Particionar empieza por mostrar la manera en como se encuentran acomodados los elementos de la lista e imprime en pantalla el primer elemento de la sub-lista en la que se encuentra (la sub-lista podría ser la lista por completo si esta función es llamada por primera vez). De igual manera, se guarda el primer elemento y su respectivo índice de la sub-lista, el índice se guarda en la variable *i*. Posteriormente, se utiliza un for para recorrer la sub-lista, desde el elemento en donde inicia hasta aquel donde termina. Cada que se recorre un elemento de esta sub-lista, se compara el primer elemento de la sub-lista con los demás. En caso de que un elemento sea menor que el primero, se incrementará en uno el valor de *i*, y se intercambiará la posición del elemento que este en esta posición con aquel que es más chico al primer elemento. Una vez finalizado este proceso, se intercambiará el primer elemento con el elemento en el índice con el valor final de la variable *i*. Este proceso se repetirá para todas las sub-listas hasta lograr que la lista por completo esté ordenada.
Para poder ordenar la lista en manera descendente, basta con cambiar la condición dentro del for en la función Particionar de esta manera: `if(A[j]>=x):`

Dada la manera en cómo se codificó, el pivote siempre será el primer elemento de cada sub-lista, pues siempre que se entra a la función Particionar, esta guarda el primer elemento de dicha.
 - b. En cuanto a HeapSort, se nos proporciona la implementación del algoritmo en Python. En esta implementación podemos observar tres funciones principales: `maxHeapify`, `construirHeapMaxIni` y `ordenacionHeapSort`.
Se empieza con a función `ordenacionHeapSort`, la cual llama a la función `construirHeapMaxIni`, con la finalidad de poder crear un max heap y, a partir de este, poder ir ordenando la lista de menor a mayor.
La función `construirHeapMaxIni` nos permite construir el max heap requerido. Para poder realizarlo, se entra a un ciclo for, el cual se repetirá desde la mitad del tamaño de la lista hasta su primer elemento. En cada iteración, se llama a la función `maxHeapify`.
La función `maxHeapify` es de tipo recursiva, pues se llamará a sí misma cuando alguno de los hijos del nodo en el que encontramos sea mayor a este, y, siempre que suceda esto, se intercambiará el nodo padre con el mayor de sus nodos hijos. Dado que el heap se guarda en una lista, todo esto se visualiza de manera teórica, en donde cada elemento se ve como un nodo, y los hijos de cada nodo se obtienen multiplicando su índice por dos y sumándole uno, para su hijo izquierdo y dos para su hijo derecho. Cabe recordar que como es un árbol binario, cada nodo puede tener de 0 a 2 hijos, inclusivamente.
El proceso anteriormente descrito se repetirá hasta haberse construido el maxHeap y se reanudará la función `ordenacionHeapSort`, la cual tiene un ciclo for. Este ciclo for se repetirá *n*-1 veces, donde *n* es el tamaño de la lista. Para cada iteración, se eliminará la raíz del max heap y este elemento se pondrá al final de la lista, pues es el mayor de todos. Como se elimina esta raíz, se coloca el último elemento del max heap en su lugar y se vuelve a llamar a la función `maxHeapify` para volver a organizar los elementos de manera que la raíz sea el mayor elemento. Es importante resaltar que, para no caer en un ciclo sin fin, el nuevo max heap se construirá sin considerar a los elementos que ya han sido eliminados.

Para poder ordenar la lista en manera descendente, basta con cambiar las condiciones dentro de la función maxHeapify de esta manera:

```
if(L<=(tamanoHeap-1) and A[L]<A[i]):  
    if(R<=(tamanoHeap-1) and A[R]<A[posMax]):
```

- c. Bubble sort consiste en ir comparando los números de la lista en parejas, cada elemento con su inmediato siguiente y en cada recorrido, mandar el número más grande de ese recorrido hasta el último.

Se nos proporcionaban dos funciones que servían para ordenar un arreglo mediante Bubble sort. La segunda función (bubbleSort2) es más eficiente, pues en caso de que se realice una pasada por el arreglo y no se cambie de lugar ningún número, entonces ya no se sigue ejecutando, pues esto nos indica que el arreglo ya se encuentra ordenado del menor al mayor valor y ya no es necesario seguirlo recorriendo más veces.

Para conocer el número de pasadas que realiza cada función se necesita de un contador, En la segunda función ya se encuentra este contador, por lo que, aquí solo es necesario imprimirlo en pantalla una vez que el algoritmo ya ordenó el arreglo. Para la primera función, se agrega una variable contadora, la cual se irá incrementando en uno cada vez que se realice el for externo.

Por último, para poder ordenar la lista del mayor al menor valor, es suficiente con cambiar la comprobación de esta manera: `if(A[j]<A[j+1]):`

Es así como, solo se cambiarán las posiciones de cada número cuando el antecesor de un número sea mayor que este.

```
[20]: def bubbleSort(A):  
        contador=0  
        for i in range(1,len(A)+1):  
            contador=contador+1  
            for j in range(len(A)-1):  
                if A[j]>A[j+1]:  
                    temp=A[j]  
                    A[j]=A[j+1]  
                    A[j+1]=temp  
            print(contador)  
  
        def bubbleSort2(A):  
            bandera=True  
            pasada=0  
            while pasada<len(A)-1 and bandera:  
                bandera=False  
                for j in range(len(A)-1-pasada):  
                    if(A[j]>A[j+1]):  
                        bandera=True  
                        temp=A[j]  
                        A[j]=A[j+1]  
                        A[j+1]=temp  
                pasada=pasada+1  
            print(pasada)  
  
        A=[10,9,16,15,7,4,5,1,13,2]  
        B=[10,9,16,15,7,4,5,1,13,2]  
        bubbleSort(B)  
        bubbleSort2(A)  
        print(A)  
        print(B)
```

10

9

[1, 2, 4, 5, 7, 9, 10, 13, 15, 16]

[1, 2, 4, 5, 7, 9, 10, 13, 15, 16]

b) Ejercicios propuestos:

- a. **Actividad 1:** Para la función de quickSort, observamos que se divide en dos funciones principales: quickSort y partition. En clase solamente se vio todo el algoritmo dentro de una misma función. La función quickSort es de tipo recursiva, pues se llama a sí misma dos veces. Siempre que el valor de low sea menor que la de high, se llamará la función a sí misma otras dos veces. Es similar a lo visto en clase, ya que se usa una comparación para saber si la función se vuelve a llamar a sí misma, siempre y cuando los valores frontera no sean los mismos o estén intercambiados. La parte del ordenamiento se realiza a través de la función partition.

La función partition nos servirá para ir ordenando el arreglo principal y, dado que quickSort utiliza un enfoque de divide y vencerás, entonces el arreglo se dividirá en sub-arreglos, los cuales también se pasarán como argumento a esta función, pues los valores frontera low y high cambiarán cuando conforme quickSort vaya llamándose a sí misma. Entrando a la función partition observamos otra diferencia con lo visto en clase, ya que en esta implementación el pivote siempre será el último elemento del sub-arreglo, mientras que en clase el pivote siempre era el elemento a la mitad del sub-arreglo.

Una vez escogido el pivote, se empezará a recorrer la sub-lista (la lista entera en caso de que se llame a la función partition por primera vez) y, cada vez que se encuentre un elemento menor o igual al pivote, se incrementará en uno la variable auxiliar i, la cual servirá para guardar el índice del primer elemento de la sub-lista y para guardar el índice del último elemento que ha sido intercambiado. Siempre que se realiza un intercambio, existen dos casos:

- i. En donde coincida el número de iteración con el valor de la variable auxiliar. En este caso el intercambio se realiza en la misma posición, es decir, el elemento se mantiene en su misma posición.
- ii. En donde el número de iteración es mayor al valor de la variable auxiliar. En este caso, el intercambio se realiza con el último elemento que es menor al pivote, lo que provocará que todos los elementos mayores al pivote queden a su derecha, y todos los elementos menores al pivote queden a su izquierda.

Es importante resaltar que el número de iteración nunca será menor al valor de la variable auxiliar, pues el valor de la variable auxiliar siempre será el valor del índice del primer elemento del sub-arreglo y el número de iteración empieza desde este valor.

Por último, en la función partition, se intercambia el elemento en la posición con el valor final de la variable auxiliar y el último elemento del sub-arreglo (el cual fue tomado como pivote), con la finalidad de posicionar al pivote en su posición correspondiente.

Las similitudes entre esta implementación y la vista en clase son las características del algoritmo quickSort, en otras palabras, lo que no cambia es el hecho de que, para ordenar el arreglo, este se divide en sub-arreglos. Cada sub-arreglo se ordena haciendo uso de un pivote, el cual tendrá en primer lugar todos los elementos mayores a este a su derecha y todos los elementos menores a este a su izquierda. Al finalizar, se posicionará al pivote en su posición correspondiente.

Para la función de bubbleSort, se agregó una variable contadora la cual siempre volverá a tener un valor de cero cuando se repita el for externo y se incrementará cuando se realice un intercambio de elementos en algún recorrido del arreglo. Si esta variable sigue teniendo su valor de cero al finalizar el recorrido del arreglo, entonces se detendrá la ejecución, pues esto quiere decir que el arreglo ya se encuentra ordenado y no es necesario seguir realizando más comparaciones.

```

        for (i=n-1; i>0; i--) {
            detenerse=0;
            for (j=0; j<i; j++) {
                if(a[j]>a[j+1]){
                    swap(&a[j],&a[j+1]);
                    detenerse++;
                }
            }
            if(detenerse==0)
                break;
        }
    }

```

Para la función de HeapSort observamos que esta se divide en tres funciones principales: HeapSort, Heapify y BuildHeap. HeapSort es la función principal, la cual comienza por llamar a la función BuildHeap, pasándole como argumentos el arreglo mediante su puntero y el tamaño de dicho.

La función BuildHeap se encargará de crear el heap de manera bottom-up, pues se creará el heap sobre la colección original. Para poder realizarlo, empezará por darle un valor a la variable heapSize, la cual es de tipo global y nos servirá para almacenar el índice del último elemento del arreglo. Una vez hecho esto, se entra a un ciclo for, el cual se repetirá desde la mitad del tamaño del arreglo hasta el primer elemento, y en cada repetición se llamará a la función Heapify, a la cual se le pasa como argumento el número de iteración (lo que representa el nodo que se acomodará) y el tamaño total del arreglo.

La función Heapify, que es de tipo recursiva ya que se llama a sí misma una vez, es la más importante, pues está se encargará de ordenar los elementos de forma que sean un max heap. Para lograrlo, almacena los índices de los hijos del nodo en el que se encuentra, los cuales se obtienen multiplicando el índice del nodo padre por dos y sumando uno para el hijo izquierdo y sumando dos para el hijo derecho. Esto se realiza debido a que el max heap se almacena en forma de arreglo. Después, comprobará primero si dicho nodo tiene hijos, y cual de dichos hijos es el mayor. El índice del que resulte ser mayor será guardado en la variable auxiliar largest. Una vez realizado esto, se procede a comparar si los hijos son mayores a su nodo padre, en caso de serlo, se intercambiará el nodo padre con su nodo hijo cuyo valor sea más grande y la función se vuelve a llamar a sí misma para posicionar a este nuevo nodo que pasó a ser hijo en el lugar donde sus hijos sean menores que este o no tenga ningún hijo.

Ya que se ha creado el heap se regresa a la función HeapSort, la cual entrará a un for y en cada iteración intercambiará el valor del último nodo del heap con la raíz y después se decrementará en uno el valor de heapSize (variable global), pues de esta manera se simula que la raíz del heap ha sido eliminada. Ya que se ha realizado esto, se vuelve a llamar a la función Heapify para volver a crear un max heap. Este proceso se repetirá hasta que solo quede un elemento en el arreglo y, dado que en cada iteración se pondrá al elemento mayor al final, el arreglo quedará ordenado.

Es importante añadir que inicialmente estas funciones solo estaban hechas para arreglos con 11 elementos únicamente, por lo que, para generalizarlo a arreglos con n elementos, se añadió un argumento a cada función para que se tome en cuenta el tamaño del arreglo.

- b. **Actividad 2:** Las capturas de implementación de los algoritmos de ordenamiento Quick Sort, Bubble Sort y Heap Sort sobre un arreglo de 10 elementos son las siguientes respectivamente:

```

El arreglo creado tiene los elementos:
26 7 9 24 9 11 1 5 15 30
***Bienvenido al programa de ordenamiento***
Seleccione el numero de la opcion que desee para ordenar el arreglo:
1)Quick Sort
2)Bubble Sort
3)Heap Sort
1
Ordenando el arreglo...
Pivote: 30    26 7 9 24 9 11 1 5 15 30
Sub array :  26 7 9 24 9 11 1 5 15
Pivote: 15    7 9 9 11 1 5 15 24 26
Sub array :   7 9 9 11 1 5
Pivote: 5     1 5 9 11 7 9
Sub array :    1
Sub array :   9 11 7 9
Pivote: 9     1 5 9 7 9 11
Sub array :    9 7
Pivote: 7     1 5 7 9
Sub array :
Sub array :   9
Sub array :  11
Sub array :  24 26
Pivote: 26    1 5 7 9 9 11 15 24 26
Sub array :   24
Sub array :
Sub array :
El arreglo ordenando es:
1 5 7 9 9 11 15 24 26 30
Program ended with exit code: 175

```

```

El arreglo creado tiene los elementos:
1 13 29 25 10 12 19 20 15 3
***Bienvenido al programa de ordenamiento***
Seleccione el numero de la opcion que desee para ordenar el arreglo:
1)Quick Sort
2)Bubble Sort
3)Heap Sort
2
Ordenando el arreglo...
1 13 25 10 12 19 20 15 3 29
1 13 10 12 19 20 15 3 25 29
1 10 12 13 19 15 3 20 25 29
1 10 12 13 15 3 19 20 25 29
1 10 12 13 3 15 19 20 25 29
1 10 12 3 13 15 19 20 25 29
1 10 3 12 13 15 19 20 25 29
1 3 10 12 13 15 19 20 25 29
1 3 10 12 13 15 19 20 25 29
El arreglo ordenando es:
1 3 10 12 13 15 19 20 25 29
Program ended with exit code: 175

```

```

El arreglo creado tiene los elementos:
16 16 16 28 22 2 5 13 30 17
***Bienvenido al programa de ordenamiento***
Seleccione el numero de la opcion que desee para ordenar el arreglo:
1)Quick Sort
2)Bubble Sort
3)Heap Sort
3
Ordenando el arreglo...
16 16 16 30 22 2 5 13 28 17
16 30 16 16 22 2 5 13 28 17
16 30 16 28 22 2 5 13 16 17
30 16 16 28 22 2 5 13 16 17
30 28 16 16 22 2 5 13 16 17
Termin\242 de construir el HEAP
Iteracion HS:
17 28 16 16 22 2 5 13 16 30
28 17 16 16 22 2 5 13 16 30
28 22 16 16 17 2 5 13 16 30

```



```

Iteracion HS:
16 22 16 16 17 2 5 13 28 30
22 16 16 16 17 2 5 13 28 30
22 17 16 16 16 2 5 13 28 30
Iteracion HS:
13 17 16 16 16 2 5 22 28 30
17 13 16 16 16 2 5 22 28 30
17 16 16 13 16 2 5 22 28 30
Iteracion HS:
5 16 16 13 16 2 17 22 28 30
16 5 16 13 16 2 17 22 28 30
16 16 16 13 5 2 17 22 28 30
Iteracion HS:
2 16 16 13 5 16 17 22 28 30
16 2 16 13 5 16 17 22 28 30
16 13 16 2 5 16 17 22 28 30

```

```

Iteracion HS:
5 13 16 2 16 16 17 22 28 30
16 13 5 2 16 16 17 22 28 30
Iteracion HS:
2 13 5 16 16 16 17 22 28 30
13 2 5 16 16 16 17 22 28 30
Iteracion HS:
5 2 13 16 16 16 17 22 28 30
Iteracion HS:
2 5 13 16 16 16 17 22 28 30
El arreglo ordenando es:
2 5 13 16 16 16 17 22 28 30
Program ended with exit code: 175

```

- c. **Actividad 3:** En esta actividad se nos pidió contabilizar las operaciones relevantes, por lo que, se contabilizaron las comparaciones, los intercambios y el total de operaciones realizadas. Las gráficas y las tablas con toda la información se encuentran en el archivo de Excel.

Es importante hacer notar que el mismo arreglo creado aleatoriamente era el que debían ordenar los tres algoritmos, lo que ofrecía una mejor comparación de cual realizaba mayor cantidad de operaciones sobre el arreglo.

Con base en dichas tablas y gráficas se puede determinar que:

- En el caso de Quick Sort, existe más variación en cuanto al número de comparaciones e intercambios que se realiza. Esto es debido al elemento que elegimos pivote, ya que si en un momento dado se escoge como pivote a un elemento con un valor intermedio entre todos los que se encuentran en el sub-arreglo, y además todos los demás elementos se encuentran bastante dispersos con valores mayores y menores, entonces se realizarán mayor número de intercambios, por lo que, dependerá mucho del elemento escogido como pivote y la distribución de los demás elementos. Es por esto por lo que observamos algunas veces que la gráfica crece un poco más rápido y en otras un poco más lento.

Para las comparaciones, estas variarán de igual forma dependiendo del elemento escogido como pivote, pues a partir de este se realizarán las divisiones del sub-arreglo, y dará origen a que existan más comparaciones si los sub-arreglos originados son de tamaños parecidos.

Por último, este algoritmo es el que cuenta con menor número de operaciones realizadas, por lo que es el más eficiente de estos algoritmos de ordenamiento.

Este algoritmo tiene el menor crecimiento entre los algoritmos de ordenamiento vistos hasta el momento, sin embargo, puede llegar a variar bastante.

- Para el caso de Bubble Sort podemos observar que de igual forma existe variación con respecto al número de intercambios realizadas. Esto es más que nada por la forma en la que se encuentran distribuidos los elementos, pues si los elementos mayores están al inicio del arreglo, se necesitarán más intercambios para mandarlos al final del arreglo. En cuanto a la cantidad de comparaciones, observamos poca variación entre los datos, pues se comparan todos los elementos en cada recorrido del arreglo, solamente varían por la mejora que se le hizo al momento de codificarlo, la cual consiste en que, si no se realiza ningún intercambio en un recorrido, entonces se termina la ejecución.

Este algoritmo tiene un crecimiento bastante rápido, por lo que estaría en la posición más baja entre todos los algoritmos de ordenamiento vistos hasta el momento.

- El caso de Heap Sort muestra poca variación en la cantidad de operaciones realizadas. Esto es debido a que el arreglo es trabajado como un max heap, dando origen a que cada elemento sea visto como un nodo y cada nodo tenga a lo más dos hijos, por lo que las operaciones realizadas solo se realizan entre el nodo padre y sus nodos hijos, además de que todas estas operaciones se empiezan a realizar de la mitad del arreglo hasta el primer elemento

La mayor cantidad de operaciones que se realizan son las comparaciones, pues primero se verifica si un nodo tiene hijos, después cual es el mayor de sus hijos, luego si alguno de ellos es mayor al padre y, en caso de serlo, comparar si este nuevo nodo hijo está en su posición correcta, de no ser así, se repetirá todo el proceso de nuevo, por lo que existe mayor cantidad de comparaciones y es lo que provoca un mayor número de operaciones totales realizadas.

Este algoritmo tiene un crecimiento bastante lineal, pues la cantidad de operaciones realizadas va creciendo de manera paulatina con la entrada recibida, sin embargo, sus operaciones totales es casi el doble que las realizadas por Quick Sort, por lo que se ubica en un nivel intermedio de los algoritmos de ordenamiento.

- d. **Actividad 4:** Para esta actividad se nos pedía realizar la implementación de Quick Sort haciendo uso del lenguaje de programación Java.

El programa es bastante parecido a la implementación realizada en C, el único cambio es que no existe una función intercambio, si no que se realiza el intercambio en el mismo lugar donde se necesita, con la ayuda de una variable auxiliar.

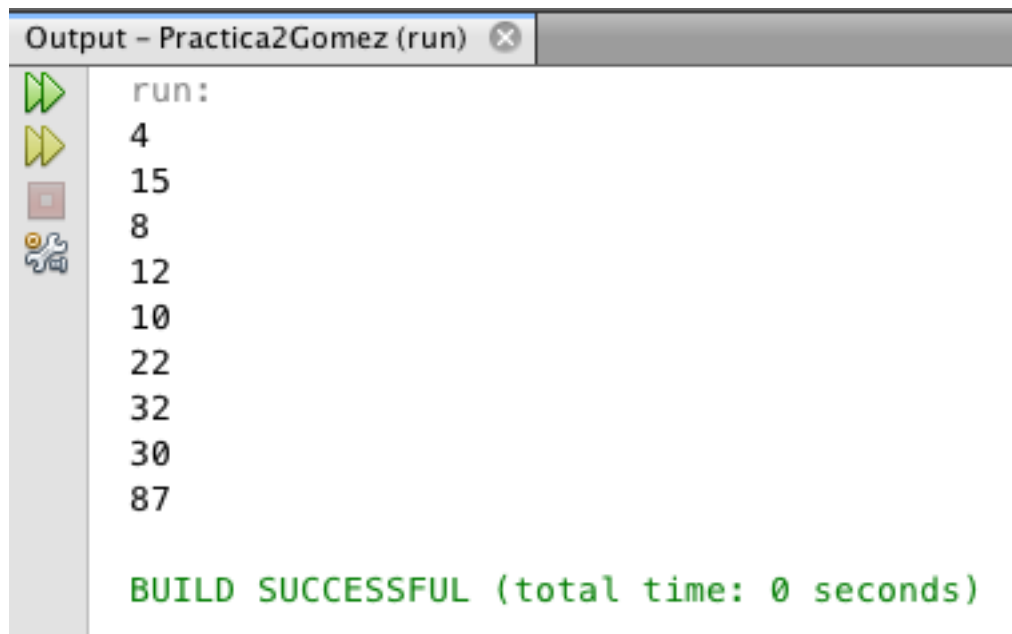
La implementación en Java se dividió en varias clases, en donde cada clase era una parte del programa. La clase que realiza el algoritmo de Quick Sort tiene el nombre de Quicksort, y tiene dos métodos: partition y sort. Estos métodos son bastante parecidos a los ya vistos en la implementación de C.

Por otro lado, tenemos la clase Utilidades, la cual tiene un solo método: printArray, la cual servirá, al igual que en la implementación en C, para imprimir el arreglo, con la particularidad de que después de cada elemento habrá un salto de línea, debido a la función System.out.println.

Por último, está la clase Ejercicio4, la cual contiene al método principal y ocupará los métodos de las demás clases realizadas con anterioridad. Es importante remarcar que, para poder usar los métodos de Quicksort, se tiene que crear una instancia de esta clase (objeto) que para este caso se nombró como: "ordenamientos". Esto se debe a que los métodos dentro de la clase de Quicksort no son estáticos, mientras que, el método dentro de Utilidades es de tipo estático, por lo tanto, no es necesario crear una instancia para usarlo, solamente con especificar el nombre de la clase, usar el operador miembro y escribir el método es suficiente para poder usarlo.

En la clase principal, se crea el arreglo que se ordenará y este se pasa como argumento al método sort, al igual que su tamaño. En esta parte podemos observar que, para conocer el tamaño de un arreglo, se utiliza el operador miembro en el arreglo que hemos creado y escribimos el nombre de su miembro length.

Lo importante en esta implementación de código son las características importantes de Java, como lo son la creación y uso de métodos estáticos y no estáticos, la creación de instancias de clase, los cuales nombrados comúnmente como objetos de una clase, entre otras características.



```
run:
4
15
8
12
10
22
32
30
87

BUILD SUCCESSFUL (total time: 0 seconds)
```

- Conclusiones

Para esta práctica se logró identificar las partes más importantes de los algoritmos de ordenamiento Bubble sort, Quick sort y Heap Sort. También se comparó la cantidad de operaciones realizadas entre cada uno de ellos, con la finalidad de determinar cual de ellos es el mejor y cual el peor. Contando estas operaciones y realizando sus gráficas correspondientes se observó que el mejor es Quick sort, pues es el que tiene menor cantidad de operaciones realizadas y el peor es Bubble Sort, pues su cantidad de operaciones crece demasiado rápido conforme aumenta el número de elementos de entrada.

La cantidad de operaciones tuvo mayor variación en Quick Sort, pues dependía de el pivote escogido en cada sub-arreglo y la dispersión de los elementos en mayores y menores.

En Bubble sort, se logró la mejora propuesta, pues si no se realizan intercambios, se detiene la ejecución. A pesar de esta pequeña mejora, este algoritmo de ordenamiento realiza gran número de operaciones, lo que lo hace más lento y de los peores algoritmos de ordenamiento. Su gran ventaja es la simpleza para comprender su funcionamiento.

En Heap sort se visualizó que la mayor cantidad de operaciones realizadas eran las comparaciones, pues los intercambios eran realizados en mucho menor proporción.

Estos algoritmos nos ofrecen diversas maneras de ordenar un arreglo y, dependiendo del arreglo que queramos ordenar, podremos escoger cual es la mejor opción. Para arreglos con pocos elementos, Bubble Sort podría ser una buena opción, pues es bastante simple. Para arreglos con una cantidad intermedia de elementos, Heap sort es la mejor opción, ya que no tiene un crecimiento tan acelerado con respecto a la cantidad de elementos proporcionados. Para arreglos con muchos elementos, Quick sort es lo recomendable, ya que es el que tiene la menor cantidad de operaciones realizadas.

El objetivo se cumplió, pues se visualizaron y analizaron las estructuras que componen a cada uno de los algoritmos de ordenamiento, además de que se compararon entre sí para determinar el mejor.

La práctica nos ofrece una buena profundización en estos algoritmos de ordenamiento, nos permite identificar sus características principales, la manera en como implementarlos en C (para el caso de Quick sort también en Java) y analizar sus cantidades de operaciones realizadas (intercambios y comparaciones). Debido a todas las actividades en la práctica se pudo lograr lo anteriormente citado, por lo que la práctica fue más que adecuada para cumplir el objetivo citado.