



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): 03

Integrante(s): GÓMEZ LUNA ALEJANDRO

*No. de Equipo de
cómputo empleado:* 47

No. de Lista o Brigada: 47

Semestre: 2020-1

Fecha de entrega: 1/Septiembre/2019

Observaciones:

CALIFICACIÓN: _____

Objetivo de la práctica

EL ESTUDIANTE IDENTIFICARÁ LA ESTRUCTURA DE LOS ALGORITMOS DE ORDENAMIENTO MERGE SORT, COUNTING SORT Y RADIX SORT.

Desarrollo

a) Ejercicios de la guía:

Se nos proporcionaban las implementaciones en Python de los algoritmos de ordenamiento de: Counting Sort, Radix Sort y Merge Sort.

Para Counting Sort, se utiliza la función `countingSort`, la cual toma como parámetros la lista a ordenar y el valor del mayor elemento en la lista. Dentro de la lista se crean dos listas auxiliares. La primera, nombrada `C`, será aquella lista en donde el valor de cada elemento será su índice en esta lista, y se irá incrementando en uno dicha posición, pues aquellos elementos con el mismo valor tendrán igual valor de índice. Una vez realizado esto para toda la lista, se empezará a realizar una suma desde el primer elemento con su inmediato anterior, así hasta llegar al último elemento. Esto se realiza con la finalidad de conocer a la posición correspondiente de cada elemento.

Por último, se acomodan los elementos en la lista `B`, con ayuda de la lista `C`, pues esta nos proporciona el índice correspondiente de cada elemento. Para aquellos elementos repetidos, una vez asignado un valor a su correspondiente índice, se decrementa en uno el índice de ese elemento en la lista `C`, pues de esa forma el próximo elemento con mismo valor se acomodará en el espacio disponible anterior.

Con este procedimiento la lista se ordena de manera ascendente, pues a la lista auxiliar `C`, cuando se recorre y se suman los valores con su anterior correspondiente, el índice que les corresponde irá creciendo, por lo que los elementos mayores se irán acomodando al final. Para poder ordenar la lista en orden inversa, basta con cambiar esta suma, para que se realice a partir del penúltimo elemento hasta el primero, como se muestra en la siguiente captura:

```
for i in range(k-1,1,-1):  
    C[i-1]=C[i-1]+C[i]
```

Por otro lado, para Radix Sort se utilizan cuatro funciones. La función principal es `radixSort`, la cual manda a llamar a la función `formarListaConClaves`, `countingSort2` y `obtenerElemSinClaves` para cada carácter del primer elemento, empezando desde el menos significativo hasta llegar al más significativo. La primera función es `formarListaConClaves`, la cual recibe como parámetros la lista a ordenar y el número de carácter actual. Este carácter irá variando desde el menos significativo hasta el más significativo, es decir, de derecha a izquierda. Debido a que se trata de una lista con caracteres y números, se realiza un mapa de bits, en el cual se almacenará el valor de cada número y carácter con su respectivo número entero que lo representa, con la finalidad de poder ir ordenando la lista mediante ese número. Una vez generado asignada la clave (numero entero) para cada elemento, se llama a la función `countingSort2`, la cual realiza un ordenamiento mediante counting sort sobre las claves obtenidas con anterioridad. Ya que se ha realizado el ordenamiento de claves, se llama a la función `obtenerElemSinClaves`, la cual obtendrá cada elemento sin su llave, pues estos ya se encuentran ordenados, por lo que no se necesitan más para esta iteración y esta lista se guardará en lugar de la lista inicial. Este proceso se repetirá hasta que se hayan recorrido todos los caracteres del primer elemento de la lista. Cabe resaltar que, como todos los elementos tienen la misma cantidad de caracteres, se pudo haber escogido cualquier elemento, sin embargo, en caso de que existan elementos con diferente cantidad de caracteres, entonces se tendrá que escoger al elemento con mayor cantidad de caracteres, y para compensar la diferencia, se pondrán ceros al inicio de cada elemento hasta que tenga la misma cantidad de caracteres que el mayor.

Para poder invertir el orden en que se ordena, es decir, que se ordene a partir del más significativo al menos significativo, es necesario cambiar el `for`, pues la variable contadora hace referencia al carácter en el que estamos, por lo que, si empezamos desde el primer elemento hasta el último, se logrará lo requerido. La captura muestra la manera en como cambiar el `for`:

```
for i in range(0,numCar+1):
```

De igual forma, para poder visualizar la manera en como se van ordenando los elementos, se imprime la lista en la función principal radixSort, dentro del ciclo for y después de llamar a las otras tres funciones, pues estas son las encargadas de ir ordenando la lista. Quedando de esta manera:

```
def radixSort(A):  
    numCar=len(A[1])  
    for i in range(numCar,0,-1):  
        cc=formaListaConClaves(A,i)  
        ordenado=countingSort2(cc,36)  
        A=obtenerElemSinClaves(ordenado)  
        print(A)  
    return A
```

En cuanto a Merge sort, este consiste en aplicar una estrategia de divide y vencerás al arreglo(lista), en donde el arreglo se dividirá en sub-arreglos hasta llegar al caso base en donde solo quede un elemento, para que a partir de este se puedan ir «juntando» los demás, comprobando cual es mayor. Al final, ya que se juntaron todos los elementos, quedará el arreglo ordenado.

Para realizar lo anteriormente descrito, se utiliza la función principal MergeSort y las funciones CrearSubArreglo y Merge. La función principal, MergeSort, es de tipo recursiva, y se llamará a si misma hasta que se llegue al caso base, en donde la sub-lista consta de un solo elemento. Una vez llegado a esto, se realizará el retroceso recursivo, en donde se irá juntando los elementos a través de la función Merge. La función Merge se encargará de ir recorriendo las dos sub-listas del retroceso recursivo en el que es llamada, con la finalidad de ir acomodando el elemento menor de cada sub-lista en la lista original.

Para poder visualizar las subsecuencias(sub-listas) obtenidas durante la ejecución, se imprime en pantalla el arreglo cada que se llama a la función CrearSubArreglo, pues esta es la encargada de realizar la partición y crear un sub-arreglo. Quedando de esta manera:

```
def CrearSubArreglo(A,indIzq,indDer):  
    print(A[indIzq:indDer+1])  
    return A[indIzq:indDer+1]
```

En cuanto a ordenar la lista de mayor a menor, es suficiente con cambiar la comprobación ubicada dentro de la función Merge de esta manera:

```
if(j>=len(Der)) or (i<len(Izq) and Izq[i]>Der[j]):
```

Solo cambiando esta instrucción permitirá que al ir combinando los sub-arreglos, se vaya acomodando primero el número más grande, y así sucesivamente hasta tener todo ordenado de mayor a menor.

b) Ejercicios propuestos:

- a. Ejercicio 1: Para este ejercicio se nos pedía implementar el algoritmo de ordenamiento Counting sort para ordenar un arreglo de caracteres, cuyo rango se encuentra entre las letras A y J en intervalo cerrado.

Para poder realizarlo, se definió la clase CountingSort con tres miembros: elementos, arregloAuxiliar y ArregloOrdenado, los cuales son arreglos. Elementos es un arreglo de tipo caracteres, que será llenada con los valores ingresados por el usuario. ArregloAuxiliar es un arreglo de enteros, el cual nos permitirá llevar la cuenta de la aparición de cada letra en el arreglo que ordenaremos, y también será en el que iremos realizando la suma para conocer la posición correspondiente de cada letra.

ArregloOrdenado es un arreglo de caracteres donde posicionaremos los caracteres de la lista original en su posición correspondiente, es decir, ya ordenados.

Estos tres arreglos se inicializan en el constructor de la clase, por lo que, se inicializarán cuando se cree una instancia de la clase CountingSort.

Una vez hecho esto, se pasa a la función principal main. En esta crearemos una instancia de la clase Scanner para recibir las letras ingresadas por el usuario y una instancia de la clase CountingSort para hacer uso de sus miembros.

Al usuario se le solicitan los caracteres que tendrá el arreglo. Ya que se ocuparon los 20 espacios del arreglo, se muestra la manera en como quedó el arreglo y se comienza a llenar el arregloAuxiliar. Cabe resaltar que todas las letras fueron inicialmente pasadas a mayúscula, por lo que su valor en ASCII empieza a partir del 65. Debido a que en el arregloAuxiliar los índices son dados por los elementos en el arreglo original, habría muchas localidades de memoria desperdiciadas, por lo que se le resta el número 65, que es el mínimo valor que puede tomar una letra en el intervalo dado, dando como resultado que se ocupen las primeras localidades de memoria sin necesidad de crear un arreglo muy grande. Ya contabilizadas todas las apariciones de cada letra, se realiza la suma en el arregloAuxiliar desde el segundo elemento con su anterior.

Luego, se acomodan los valores en arregloOrdenando, partiendo desde el final de la lista original y como el índice de cada letra es determinado por su valor, se le tiene que restar 65 por lo explicado anteriormente. Es importante mencionar que en ambos casos cuando se realiza la resta, se aplica un cast hacía int, con la finalidad de no tener conflicto con utilizar el resultado como índice, ya que el índice es un número entero.

Por último, se muestra el arreglo ya ordenado.

```
Granada47:Desktop alumno$ java CountingSort
Ingrese una letra en el rango A-J:
b
Ingrese una letra en el rango A-J:
b
Ingrese una letra en el rango A-J:
a
Ingrese una letra en el rango A-J:
a
Ingrese una letra en el rango A-J:
a
Ingrese una letra en el rango A-J:
c
Ingrese una letra en el rango A-J:
c
Ingrese una letra en el rango A-J:
c
Ingrese una letra en el rango A-J:
c
Ingrese una letra en el rango A-J:
j
Ingrese una letra en el rango A-J:
j
Ingrese una letra en el rango A-J:
e
Ingrese una letra en el rango A-J:
f
Ingrese una letra en el rango A-J:
f
Ingrese una letra en el rango A-J:
f
Ingrese una letra en el rango A-J:
h
Ingrese una letra en el rango A-J:
d
Ingrese una letra en el rango A-J:
g
Ingrese una letra en el rango A-J:
i
Ingrese una letra en el rango A-J:
i
Arreglo inicial:
B B A A A C C C C J J E F F F H D G I I
3 5 9 10 11 14 15 16 18 20 20 20 20 20 20 20 20 20 20 20
      I
      I I
    G I I
  D G I I
D G H I I
D F G H I I
D F F G H I I
D F F F G H I I
D E F F F G H I I
D E F F F G H I I J
D E F F F G H I I J J
C D E F F F G H I I J J
C C D E F F F G H I I J J
C C C D E F F F G H I I J J
A C C C C D E F F F G H I I J J
A A C C C C D E F F F G H I I J J
A A A C C C C D E F F F G H I I J J
A A A B C C C C D E F F F G H I I J J
A A A B B C C C C D E F F F G H I I J J
Arreglo final ordenado:
A A A B B C C C C D E F F F G H I I J J
```

- b. Ejercicio 2: Para este ejercicio se nos pedía implementar el algoritmo de ordenamiento Radix Sort para ordenar una lista de elementos cuyos dígitos varían entre 0 y 4 intervalo cerrado.

Para realizarlo se empezó por importar las bibliotecas de java que nos permiten trabajar con el Scanner, lista ligada, cola y arraylist. Luego se creó la clase RadixSort, la cual cuenta con un arraylist y cinco colas como atributos. Todos estos miembros se inicializan en el constructor de la clase. Se añade que, la cola es una interfaz, por lo que es de tipo abstracto y no se puede inicializar, entonces se inicializa como si fuera una lista ligada.

Además del método, está el método radixSort, el cual implementará el ordenamiento.

Empezamos por obtener la longitud del primer dígito en la lista, pues esto nos permitirá saber la cantidad de repeticiones que haremos en total y para cada elemento.

Con este dato, creamos un for externo que se repetirá la cantidad de dígitos que tenga el primer elemento, pues se parte de que todos los elementos tienen la misma cantidad de dígitos. Dentro de este for, existe otro for, el for interno, el cual obtendrá el dígito del número con el que estamos trabajando en ese momento y, dependiendo de dicho dígito, lo colocará en una cola. Ya que se realizó esto para todos los elementos, se procede a desencolar todos los elementos de cada cola, empezando por la cola que corresponde al dígito 0 y terminando con la cola que corresponde al dígito 4, pues se quiere que los elementos mayores con dígito mayor en ese momento se posicionen al último y aquellos con dígito menor al principio. Este proceso se repetirá hasta que el for externo finalice.

Por último, se crea la función principal main, donde existirá un pequeño menú para que el usuario seleccione si quiere meter más elementos, ordenar los elementos actuales o salir. En caso de ordenar los elementos, se muestra la lista original y la lista ya ordenada.

Debido a que no se especifica el número máximo de elementos que se ordenará, entonces se utiliza un arrayList, pues es un arreglo de tipo dinámico (uso de memoria dinámica).

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
1011
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
1201
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
2113
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
3130
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
3311
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
4321
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
2340
```

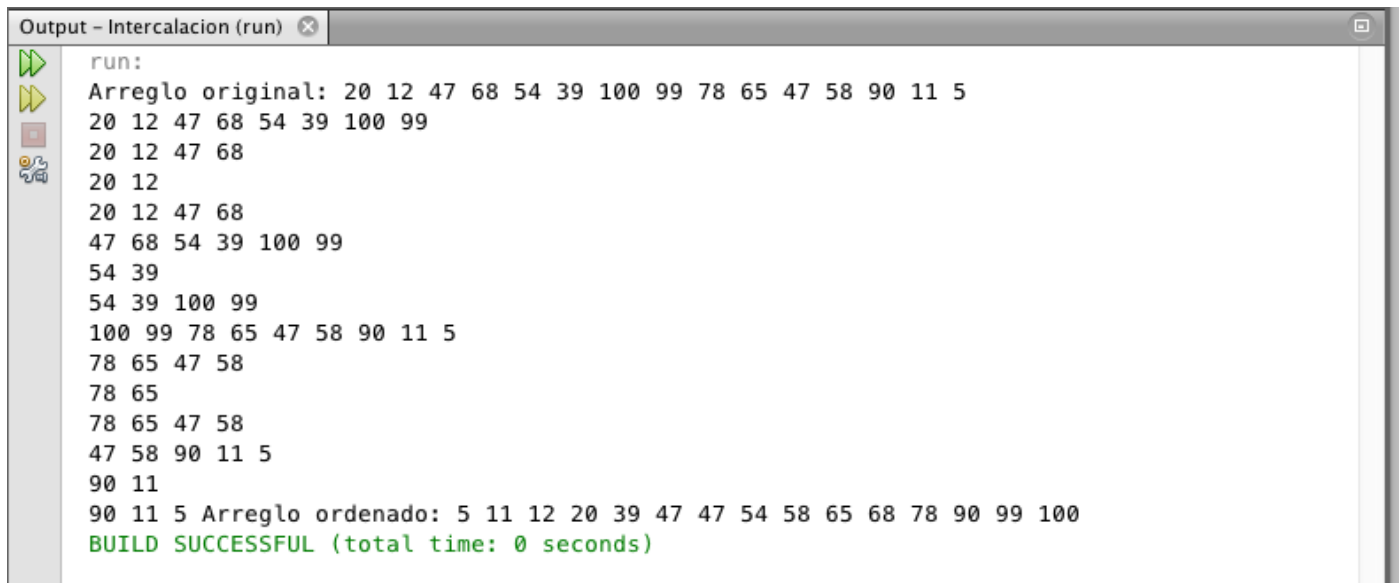
```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
2
Arreglo inicial: 1011 1201 2113 3130 3311 4321 2340
Valores de la lista: 3130 2340 1011 1201 3311 4321 2113
Valores de la lista: 1201 1011 3311 2113 4321 3130 2340
Valores de la lista: 1011 2113 3130 1201 3311 4321 2340
Valores de la lista: 1011 1201 2113 2340 3130 3311 4321
Arreglo ordenado: 1011 1201 2113 2340 3130 3311 4321
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
1
Ingrese un numero de cuatro digitos:
1231
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
2
Arreglo inicial: 1011 1201 2113 2340 3130 3311 4321 1231
Valores de la lista: 2340 3130 1011 1201 3311 4321 1231 2113
Valores de la lista: 1201 1011 3311 2113 4321 3130 1231 2340
Valores de la lista: 1011 2113 3130 1201 1231 3311 4321 2340
Valores de la lista: 1011 1201 1231 2113 2340 3130 3311 4321
Arreglo ordenado: 1011 1201 1231 2113 2340 3130 3311 4321
```

```
***RADIX SORT***
Seleccione la opcion que desea:
1)Insertar un nuevo elemento
2)Ordenar los elementos
3)Salir
3
```


- c. Ejercicio 3: Para este ejercicio se nos proporcionó la clase MergeSort, la cual cuenta con los métodos printArray, merge y sort. Los métodos merge y sort van a ser los encargados de ordenar la lista en dos fases. La primera fase consiste en ir dividiendo la lista a la mitad hasta llegar al caso en donde solo nos queden dos elementos. Esta fase es realizada por sort y se realiza mediante la recursividad, pues la función se llama a sí mismo para ir particionado a la lista. La lista se parte por la mitad, y para los números impares, solo se toma la parte entera de la división entre dos realizada para determinar la mitad.
- Cuando se llega al caso base, en el retroceso recursivo es cuando se empiezan a juntar las sub-listas creadas. Para juntarlas se hace uso de la función merge. Esta función merge tomará como parámetro el inicio y fin de la sub-lista en la que ha sido llamada, así como su punto medio. Se declaran dos variables, n1 y n2. Estas variables almacenarán el valor de hasta que punto llega el lado izquierdo de la sub-lista y hasta punto llegar el lado derecho de la sub-lista respectivamente. Para el caso izquierdo, el punto hasta donde llega es el punto medio, incluyéndolo. Esto se realiza con la finalidad de crear dos arreglos, uno para la parte derecha y otro para la parte izquierda, los cuales se llenarán con dichos elementos.
- Con estos arreglos ya llenados, se empiezan a comparar los elementos de cada arreglo (izquierdo y derecho), con la finalidad de poder ir posicionando cada elemento en su posición correspondiente en el arreglo original. Si alguno de los dos arreglos se vacía primero que el otro, esto nos indicará que el arreglo resultante solo contendrá elementos mayores, por lo que ya no habrá que realizar más comparaciones y solo terminar de llenar el arreglo original con los elementos del arreglo resultante.



```
run:
Arreglo original: 20 12 47 68 54 39 100 99 78 65 47 58 90 11 5
20 12 47 68 54 39 100 99
20 12 47 68
20 12
20 12 47 68
47 68 54 39 100 99
54 39
54 39 100 99
100 99 78 65 47 58 90 11 5
78 65 47 58
78 65
78 65 47 58
47 58 90 11 5
90 11
90 11 5 Arreglo ordenado: 5 11 12 20 39 47 47 54 58 65 68 78 90 99 100
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusiones

En esta práctica se logró implementar los algoritmos de Counting Sort y de Radix Sort, así como el poder analizar la estructura del algoritmo de ordenamiento Merge Sort. Los primeros dos ejercicios pedían implementar Counting Sort y Radix Sort, aunque el pseudocódigo de estos dos algoritmos no se vio en clase, solamente en la guía venía una pequeña implementación en Python de cada uno, por lo que existía un grado de complejidad un tanto mayor. Aunada a esta dificultad, se encontraba el hecho de que, para el primer ejercicio, se tenía que realizar Counting Sort sobre una lista de caracteres, por lo que existía una complejidad aun mayor, pues había que asociar cada letra a un número, contar la aparición de cada letra, entre otras cuestiones. En cuanto al segundo ejercicio, se tenía la dificultad de ir acomodando del carácter menos significativo al más significativo para cada elemento, además de que, conforme al número que se tuviera, acomodarlo en una respectiva cola.

A pesar de las dificultades presentadas en esta práctica, se pudieron resolver las actividades y realizar la implementación de los algoritmos de ordenamiento. Para todas las implementaciones se utilizó el lenguaje de programación Java, pues ofrece herramientas bastante útiles que faciliten el manejo de los datos, como es el caso de las clases ya definidas por Java, las cuales nos permiten implementar diversas colecciones, junto con sus respectivos métodos.

Estos algoritmos de ordenamiento son bastante eficaces, pues pueden organizar bastantes elementos en un tiempo bastante corto, sin embargo, cada algoritmo presenta desventajas:

- Counting Sort necesita bastante memoria adicional si los elementos a ordenar tienen un rango de valores bastante amplio, por lo que se empezaría crear un desbalance entre el uso de memoria y el tiempo requerido.
- Radix Sort también necesita bastante memoria adicional si los elementos a ordenar tienen dígitos bastante diferentes, pues se tiene que crear una cola para cada dígito. Además, los elementos deben tener una cantidad de dígitos bastante parecida.
- Merge Sort necesita memoria adicional, pero llega a ser mucho menor que la usada por los otros dos algoritmos de ordenamiento. Aun así, conforme se tengan muchos más elementos en la lista, se empezará a notar un uso de espacio bastante considerable.

La práctica fue un tanto difícil, ya que no se nos daban las implementaciones de los algoritmos de ordenamiento, a excepción de Merge, por lo que requería de mayor esfuerzo por parte nuestra para poder pensar en como realizar la implementación. A pesar de la dificultad, la práctica cumple con su propósito, porque, junto con los ejemplos de la guía, podemos observar diversas maneras en como podemos implementar estos algoritmos de ordenamiento, apreciar mejor el funcionamiento que tienen. En el caso de Counting Sort, el hecho de aplicar un ordenamiento a una lista de caracteres permite apreciar de mejor manera su funcionamiento.

Llegados a este punto, hemos visto la estructura y el funcionamiento de los algoritmos de ordenamiento principales. Cada uno de los algoritmos tiene sus ventajas y desventajas. Entre las ventajas encontradas existen algoritmos que son simples de implementar, sirven para listas con muchos elementos, son bastante rápidos, entre otras ventajas. Entre las desventajas encontradas existen algoritmos que son difíciles de implementar, son bastante lentos, solo sirven para listas con pocos elementos, utilizan mucha memoria adicional, entre otras desventajas.

De igual manera, comparamos los algoritmos de ordenamiento mediante la cantidad de intercambios y comparaciones que realizaban, lo que nos daba una mejor idea de cuales algoritmos eran mejores. Tomando todo lo anteriormente dicho en cuenta, podemos elegir que algoritmo de ordenamiento elegir. Si requerimos ordenar una lista con muy pocos elementos podríamos escoger un ordenamiento por burbuja, pues es bastante simple para implementar. Si requerimos ordenar una lista con pocos elementos podríamos escoger Insertion Sort o Selection Sort, pues de igual manera son simples de implementar y ofrecer un mejor tiempo que ordenamiento por burbuja. Si requerimos ordenar una lista con bastantes elementos, Heap Sort es buena opción, pues tiene un tiempo de complejidad bastante aceptable para una cantidad de elementos grande. Si requerimos ordenar una lista con muchísimos elementos, Quick Sort es la mejor opción, pues es el que tiene mejor tiempo de complejidad. También se podría ocupar Merge Sort, pero se tendría que considerar si queremos o no utilizar memoria adicional. Si requerimos ordenar una lista con elementos dentro de un rango de valores estrecho, Counting Sort permitiría hacerlo de una manera bastante eficaz, sin sacrificar tanta memoria debido a que el rango de valores no es tan amplio, Si tenemos una lista con elementos cuyos dígitos y número de dígitos son parecidos, entonces podríamos implementar Radix Sort.

En conclusión, con todo nuestro bagaje en algoritmos de ordenamiento, podremos escoger el algoritmo adecuado para distintas entradas de elementos.