**Justificación Taller 6 de Ordenación.**

**Myllee Sarleth Mosquera Rivas**

**202010024101**

**2.** La eficiencia de estos 2 métodos es la misma, ya que se implementa el mismo código, y la estrategia que se usa en él permite que se tenga la misma complejidad en cualquiera de los casos. Como vemos el método hace uso de el not in, el cual permite identificar si un elemento pertenece o no a una lista, y en su proceso recorre toda la lista para tener constancia de que el elemento pertenece o no a la lista, como vemos también, se hace uso de la función remove, que elimina todos los elementos que sean igual al que se le entregó.

La complejidad del método es de O(n), por lo que se puede decir que es eficiente, ya que tiene una complejidad baja.

**3.** Puede estarse usando tanto ordenamiento Burbuja como Inserción debido a que en cada uno se hace lo siguiente:

* **Burbuja:** en la primera pasada toma el primer valor de la lista que en este caso es el 47, y pregunta ¿47>3?, como la respuesta es sí los intercambia, quedando así: [3,47,21,32,56,92]

Posteriormente evalúa si el número que se encuentra después del 47 es mayor que él, por lo que pregunta ¿47>21?, al ser cierto se debe intercambiar quedando así: [3,21,47,32,56,92]

* **Inserción:** En este se hace una comparación en cuanto al número que se encuentra a la izquierda del elemento y no a la derecha como lo hacemos en burbuja, como a la izquierda del 47 no hay ningún elemento debido a que es el primer elemento de la lista, pasamos a ver el siguiente, entonces se pregunta ¿3<47? Como la respuesta es sí se intercambian, quedando así [3,47,21,32,56,92]

En la siguiente pasada tiene como objeto a intercambiar el 21, ya que el 3 y el 47 están ordenados, pregunta ¿21<47? Como la respuesta es sí, se intercambian quedando así: [3,21,47,32,56,92]

**4**. Se hacen 12 intercambios y 3 pasadas, esta cantidad de pasadas se debe a que son 10 elementos por lo que se divide 10 entre 2, dando la instrucción de que la resta de las posiciones de los números a comparar en la primera pasada debe ser 5, se empiezan a hacer las comparaciones según las posiciones y los intercambios necesarios, en la primera pasada se hacen comparaciones de tal forma que entre los 2 números que se vayan a comparar hayan 4 números, en la pasada 2, deberán haber 2 entre ellos, ya que se divide 4 entre 2, en la siguiente pasada debe haber un número o elemento entre ellos y en la última pasada no debe haber ninguno entre ellos, por tanto se realizan 3 pasadas en total.

**5.** Complejidad del algoritmo es de O(n), ya que contiene un for que tiene complejidad O(n), y contiene una serie de instrucciones de orden O (1), por lo que al realizar el producto de éstos se obtiene que la complejidad del código es de O(n)

**6**.

1. Al aplicar el método sort se ordena lisa de tuplas futbolistasTup.
2. En el parámetro (key=lambda futbolista: futbolista[0]), se está especificando que queremos que la lista de tuplas sea ordenada de acuerdo al primer elemento de la tupla, que es en este caso el número, ya que este se encuentra en la posición 0 de la tupla, si la posición la cambiamos por 1, se ordenará de acuerdo al nombre del futbolista.
3. Si se aplica el método sort con parámetros a las listas de 1,3, y 4 nos arroja error, al parecer este método con parámetros solo puede ser utilizado para listas de tuplas, si se desean ordenar las listas que tenemos en los ejercicios 1,3, y 4, debemos usar el método sort sin parámetros.
4. InventosTup = [(87, "KaiOs"), (85,"Osso\_VR"), (91,"Roybi\_Robot"), (75,"CareOS"), (90,"BML100PI"), (98,"Rakuten")]

**7.** El peor de los casos sería que todos los números de la lista sean negativos, debido a que la cantidad de veces que se hace uso de la función Append() depende de la cantidad de números negativos de la lista, lo cual nos da entender que, si se tienen más números negativos, la función deberá realizar un mayor esfuerzo a comparación de lo que se haría con una lista con más números positivos.

El tiempo de ejecución de este caso sería mayor a los del caso promedio y el mejor de los casos, el tiempo de ejecución en este caso estaría dado por:

* T\_Peor\_Caso = T(for) + T(if) + T (Append ())
* T\_Mejor\_Caso=T(for) + T(if)

Es decir, el tiempo que se tarda en cumplir cada instrucción, por lo que podemos afirmar que el Tiempo en el peor de los casos es mayor ya que cumple una instrucción de más, esto dependiendo de la cantidad de números negativos.

**8.** Luego de 3 llamadas recursivas la lista queda así: [1,2,9,21,26,28,29,45], porque antes de empezar las llamadas recursivas la lista se divide en 2, de tal forma que tendríamos 2 listas:

Izquierda=[21,1,26,45,29,28,2,9]

Derecha= [16,49,39,27,43,34,46,40]

Al hacer la primera llamada recursiva se toma la lista izquierda y se divide a la mitad:

[21,1,26,45] [29,28,2,9]

Luego se hace otra llamada recursiva, por lo que se volvería a dividir en 2, así:

[21,1] [26,45] [29,28] [2,9]

Luego se hace la tercera llamada recursiva, en esta se vuelve a dividir en 2, quedando así:

[21] [1] [26] [45] [29] [28] [2] [9]

Posteriormente las listas empezarían a ordenarse y mezclarse en una sola, quedando así:

[1,2,9,21,26,28,29,45]

Como hace falta ordenar el lado derecho de la lista aún no serán mezcladas, harían falta otras 3 llamadas recursivas para la lista derecha.

**9.**

* La función **init** crea una lista vacía.
* El método **Len** nos devuelve la cantidad de elementos de la lista, esto lo hace gracias a la función **len** de Python.
* El método **Contains**, este método evalúa si un elemento pertenece a la lista, para esto realiza una llamada a la función **findPosition** que le proporciona la posición del elemento dentro de la lista, esta posición es almacenada en la variable **ndx,** para determinar si ese elemento pertenece o no, se evalúa si aquella posición es menor a la longitud de la lista y si es así, se busca el elemento que hay en esa posición para compararlo con el que es buscado.
* La función **add**, agrega un nuevo elemento a una lista, para esto se asegura de que el elemento no se encuentre en la lista, si en este no se encuentra se llamará a la función **findPosition** para conocer su posición, y ser agregado a la lista vacía que había sido creada, y en la posición arrojada por la función **findPosition**.
* La función **remove**, elimina elementos de la lista, para esto se debe estar seguro de que el elemento está en la lista, si lo está, se llama a la función **findPosition** con el mismo objetivo anteriormente mencionado, al ser conocida la posición se hace uso de la función pop de Python que permite eliminar el elemento de la lista en una determinada posición.
* La función **isSubestOf**, determina si 2 listas son iguales o si una es sublista de otra, para esto se hace uso del **not in**, que determina si un elemento pertenece o no a una lista.
* El método **iter**, permite recorrer la lista de elementos.
* La función **findPosition** encuentra la posición de un elemento dentro de una lista.
* La complejidad de este algoritmo es de O(n log n) debido a que cuenta con funciones de complejidad O(1), un for que es de O(n) y un while que sería de O(log n), por lo que al realizar el producto de todas estas nos encontramos con una complejidad de O(n logn).

**12.**

Ordenamiento **Radix Sort**

Este es un método de ordenamiento de lista, donde su principal punto de evaluación son los dígitos de cada uno de los números pertenecientes a una lista o arreglo.

Su comportamiento sería algo así:

Consideremos la siguiente lista:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 65 | 43 | 21 | 6 | 12 |

Tendremos además de esta lista, una lista de longitud 10, donde se irán almacenando los elementos de la lista dependiendo de los dígitos que contenga el número. Por ejemplo, se empieza evaluando el número 65 ya que es el primer elemento de la lista, vemos que el ultimo digito del 65 es el 5, por lo que este número se agregará en la posición 5 de la lista, lo mismo se hará con los demás elementos, de tal forma que en una primera pasada la lista auxiliar quedará así:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 21 | 12 | 43 |  | 65 | 6 |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

La razón por la cual no aparecen elementos en algunas casillas es porque ningún elemento de la lista finaliza en aquel dígito. La lista original quedaría así: [21,12,43,65,6]. Cabe aclarar que el 6, es considerado como 06, por lo que en la siguiente pasada quedará agregado en la posición 0 de la lista auxiliar, quedando así:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6 | 12 | 21 |  | 43 |  | 65 |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Finalmente, la lista quedaría así: [6,12,21,43,65]

**Bin Sort:**

Este método de ordenamiento consiste en ordenar los elementos de una lista por medio de “cajas”, en las cuales se ingresarán elementos de la lista que cumplen unas determinadas características para posteriormente organizarlas. Para determinar en qué intervalos deben estar los elementos de la caja se deben ver la cantidad de elementos de la lista y el número mayor, por ejemplo,

Considerando la lista: [8,2,3,6,7,12,23,35,40,13]

|  |  |  |  |
| --- | --- | --- | --- |
| 8,2,3,6,7 | 12,13 | 23 | 35,40 |
| 0-10 | 11-21 | 22-32 | 33-43 |

En este caso los elementos quedaron divididos en 4 cajas, con rangos de 10, puesto que son 10 elementos y el mayor elemento de la lista es el 40, así todos los elementos pertenecerán a una caja, ya que cumplen las condiciones de alguna de ellas.

Cómo podemos observar los elementos de la primera caja no están ordenados, por lo cual se debe recurrir a cualquier otro método para poder ordenarlos.