Documentación taller 6

Dónovan Castrillón Franco – 202010034010

**1)**

El código se encuentra en el archivo “punto\_1.py”

**2)**

Comparando la eficiencia de este método con el del punto anterior, nos podemos dar cuenta que este es mucho más eficiente ya que para cada nuevo número no tendremos que recorrer toda la lista en busca de alguno repetido, si no que ya sabemos que si aparece un numero diferente ya no existirán otros repetidos, simplificando el método en cuestión. Hablando del código se puede evidenciar la reducción de la complejidad en la implementación del continue.

El código, calculo de la complejidad y otras anotaciones se encuentran en el archivo “punto\_2.py”

**3)**

El algoritmo de ordenación que se uso fue el de inserción, ya que como se demostró con el algoritmo, que limitamos a únicamente 2 pasadas, queda de la manera en la que se nos dice: [3, 21, 47, 32, 56, 92], demostrando con esto la razón por la cual el de inserción fue la que se usó.

El código en cuestión se encuentra en el archivo “punto\_3.py”

**4)**

Como se demuestra en el código que se encuentra en el archivo “punto\_4.py” se termina en 3 pasadas y los intercambios específicos también se pueden evidenciar en la salida de la ejecución de dicho código.

**6)**

El código de este punto se encuentra en el archivo “punto\_6.py”

1. La lista queda ordenada según el primer dato de la tupla, refiriéndose en este caso al numero
2. Lambda es una manera simple de declarar una función, de la manera: "lambda: parámetros: expresión". Sort es un método con el cual podemos organizar listas. En este caso que tenemos una lista de tuplas la cual podríamos organizarla tomando en cuenta o el primer dato o el segundo, ahora ¿qué es lo que nos está queriendo decir "key=lambda futbolista: futbolista[0]”? Desglosémosla: "key="lo que nos quiere decir es que lo que sigue es la manera o la regla que debe seguir para organizar la lista, después lo que se hace es definir una función lambda con un parámetro llamado futbolista y luego, le sigue la función que en este caso nos esta retornando el dato en la primera posición de la tupla la cual será la que tomaremos para organizar.
3. De lo anterior podemos concluir que: si ponemos la key exactamente igual que como lo hicimos con la lista de tuplas nos dará un traceback con el siguiente error: *"'int' object is not subscriptable"* y esto se debe a que un no hay manera de iterar sobre un numero en cambio si podemos iterar o movernos sobre una tupla. Luego si cambiamos la key a una función que retorne el mismo argumento ingresado si organizará la lista normalmente pero realmente funcionaria igual si simplemente no agregamos una key o algún parámetro a los argumentos de sort, seria código innecesario.
4. Se encuentra en el código.

**7)**

En el peor de los casos se va a tener que insertar en la nueva lista todos los elementos del anterior pero siempre va a tener que recorrer toda la lista para comprobar si es negativo, y el tiempo de ejecución dependerá de la cantidad de elementos que tenga la lista inicial y de la cantidad de elementos a insertar en la nueva

El código de este punto se encuentra en el archivo “punto\_7.py”

**8)**

Después de 3 llamadas recursivas la lista estaría igual, ya que lo hacen las llamadas recursivas es dividirla en sublistas. Así que el único “cambio” del que podríamos hablar seria que se dividió en sublistas

**9)**



\*class set = Se está creando una nueva clase con el nombre “set”

\*def \_\_init\_\_(self) = Es el método constructor de la clase cuyo único atributo es “theElements” la cual es una lista

Complejidad: O(1)

\*def \_\_len\_\_(self) = Es una función de la clase set que nos retorna la longitud del set

Complejidad: O(1)

\*def \_\_contains\_\_ (self, element) = Es una función que nos determina si el “element” en el argumento que se haya puesto existe dentro del set

Complejidad: O(1)

\*def add(self, element) = Añade un nuevo único elemento en el set, si ese elemento ya existe dentro, no lo agrega.

Complejidad: O(1)

\*def remove (self, element) = Elimina un elemento del set, este elemento claramente debe existir dentro del mismo

Complejidad: O(1)

\*def inSubsetOf (self, setB) = Determina si un set está incluido dentro de otro set

Complejidad: O(n^2)

\*def \_\_iter\_\_(self) =

Complejidad: O(1)

\*def \_findPosition(self, element) = Determina la posición del elemento que se ingresó

Complejidad: O(n)

**11)**

El código de este punto se encuentra en el archivo “punto\_11.py”

**12)**

**RadixSort:**

Imaginemos que se desea organizar esta lista en un orden ascendente:

[93, 29, 250, 86, 743, 333]

El primero paso para organizar esta lista por RadixSort es utilizar antes el CountingSort la cual significa que tomaremos el ultimo digito de cada numero para organizarla:

[93, 29, 250, 86, 743, 333]

Y quedaría así:

[250, 93, 743, 333, 86, 29]

Téngase en cuenta que como en este caso cuando hay diferentes numero con el mismo digito estos quedarán en el mismo orden en el que se encuentran.

Luego aplicaríamos el mismo CountingSort solo que ahora con el segundo digito de cada número:

[250, 93, 743, 333, 86, 29]

Y quedaría así:

[29, 333, 743, 250, 86, 93]

Luego nuevamente haremos lo mismo con el siguiente digito de los numero, y como podemos notar hay algunos numero que no tienen digito en esa posición, en esos casos se toma como si hubiera un 0:

[029, 333, 743, 250, 086, 093]

Y quedaría así:

[29, 86, 93, 250, 333, 743]

Como podemos ver ahora si la lista finalmente está organizada.

Complejidad:

O ( d ( n + b ) )

Siendo d el número de dígitos que tenga el número más grande, b siendo el número de “keys” en este caso los dígitos que son del 1 al 9, y n la cantidad de elementos en la lista.

**BucketSort o BinSort:**

Imaginemos que se quiere organizar la siguiente lista:

[28, 9, 19, 29, 15, 22, 4, 18, 7]

El numero total de elementos en la lista podemos ver que es de 9, miramos cual es el numero mayor, en este caso el 29, por lo cual Crearemos 10 posiciones de 3 diferentes rangos

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] 0 - 9

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] 10 - 19

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] 20 – 29

Luego insertamos según su rango:

[9, 4, 7] 0 - 9

[19, 15, 18] 10 - 19

[28, 29, 22] 20 – 29

Ordenamos individualmente cada uno, lo haremos por el método de inserción

[4, 7, 9]

[15, 18, 19]

[22, 28, 29]

Como siguiente solo nos queda unir nuevamente las sublistas que creamos:

[4, 7, 9] + [15, 18, 19] + [22, 28, 29]

[4, 7, 9, 15, 18, 19, 22, 28, 29]

Y como nos podemos dar cuenta la lista ya quedaría organizada.