**Resumen**

Una de las aplicaciones más frecuentes, en programación es la ordenación.

• Existen dos técnicas de ordenación fundamentales en gestión de datos: ordenación de listas y ordenación de archivos.

• Los datos se pueden ordenar en orden ascendente o en orden descendente.

• Cada recorrido de los datos durante el proceso de ordenación se conoce como pasada o iteración.

• Los algoritmos de ordenación básicos son:

• Selección.

• Inserción.

• Burbuja.

• Los algoritmos de ordenación más avanzados son:

• Shell.

• Mergesort.

• Quicksort.

• La eficiencia de los algoritmos de burbuja, inserción y selección es 0(n2).

• La eficiencia de los algoritmos heapsort, radixsort, mergesort y quicksort es 0(n Log n).

• La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista. • Existen dos métodos básicos de búsqueda en arrays: búsqueda secuencial y binaria.

• La búsqueda secuencial se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado y se llega al final de la lista.

• Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado búsqueda binaria.

• La eficiencia de una búsqueda secuencial es 0(n).

• La eficiencia de una búsqueda binaria es 0(log n).

**Taller – Harold González 202010016101**

1. (10) Se desea eliminar todos los números duplicados de una lista.

Por ejemplo:

Si se le ingresan los valores [4, 7, 11, 4, 9, 5, 11, 7, 3, 5]

Se debe cambia a [4, 7, 11, 9, 5, 3]

**R:** Revisar el archivo 1\_borrar\_duplicados.py

1. (10) Elimine los elementos duplicados de un vector ordenado.

**R:** Revisar el archivo 2\_borrar\_orden.py

¿Cuál es la eficiencia del método?

**R:** La línea 5 corresponde a 1 asignación **(1 OE).**

La línea 6 corresponde a un ciclo while que ingresa al vector y realiza una comparación **(2 OE).**

La línea 7 corresponde a un condicional if que ingresa 2 veces al vector, y realiza una comparación **(3 OE).**

La línea 8 corresponde a un acceso al vector. **(1 OE).**

La línea 10 corresponde a un incremento **(2 OE).**

La línea 11 corresponde a un retorno **(1 OE).**

Para calcular la eficiencia, debemos revisar la función del tiempo en el peor caso, que se daría cuando todos los elementos son repetidos. Así:

-Se realiza una asignación **(+1)**

-Se entra al while n veces, siendo que será necesario eliminar los n – 1 elementos repetidos y romper el ciclo **(+2n)**

-Se entra al condicional n – 1 veces, que corresponde a la cantidad de comparaciones en que el while decide continuar **(+3(n-1))**

-Siendo que el valor siguiente siempre va a ser un duplicado, la línea 8 se ejecuta n – 1 veces **(+(n-1))**

-La línea 10 no se ejecutaría en ningún momento **(+0)**.

-Se realiza el retorno **(+1)**.

Obtenemos la siguiente ecuación:

Con lo que decimos que la eficiencia es de orden lineal, es decir que tiene complejidad

Compárela con la eficiencia del punto 1.

**R:** La eficiencia del primer método es de orden cuadrático, es decir , lo cual se puede deducir porque el método utiliza ciclos anidados y la búsqueda secuencial para encontrar los índices, lo que en el peor caso resulta en realizar n veces, n pasos.

Así, podemos afirmar que la eficiencia del segundo método es mucho mejor que la del primero al realizar una menor cantidad de pasos según el número n de datos. Claro, que si en primer momento nos entregan la lista desordenada, entonces el segundo método debe recurrir a un algoritmo de ordenación y al final su eficiencia sería de , aunque sigue siendo mejor que el primer método.

1. (5) Dada la siguiente lista:

[47, 3, 21, 32, 56, 92]

Después de 2 “pasadas” de un algoritmo de ordenación, la lista ha quedado dispuesto así:

[3, 21, 47, 32, 56, 92]

¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)?

**R:** Es más probable que el algoritmo de ordenación que se utilizó es el de **selección**. Esto teniendo en cuenta que una “pasada” recorre todos los elementos. El método de inserción, en su segunda iteración también termina en esa lista, sin embargo, no cumple con la definición propuesta de “pasada”. A continuación, mostraré ambos algoritmos.

Justifique su respuesta.

**R: Selección.**

**Momento inicial:** [47, 3, 21, 32, 56, 92]

**Pasada 1 (intercambio del menor con el primer elemento):** [3, 47, 21, 32, 56, 92]

**Pasada 2 (intercambio del menor de la lista restante con el segundo elemento):** [3, 21, 47, 32, 56, 92]

Que era a lo que queríamos llegar.

**Inserción.**

**Momento inicial:** [47, 3, 21, 32, 56, 92]

**Pasada 1 (Se inserta el segundo elemento en la primera posición, se mueve el primer elemento a la derecha):** [3, 47, 21, 32, 56, 92]

**Pasada 2 (Se inserta el tercer elemento en la segunda posición, se mueve el segundo elemento a la derecha):** [3, 21, 47, 32, 56, 92]

Que era a lo que queríamos llegar.

1. (10) Utilizar el algoritmo de ordenación SHELL, encuentre las pasadas e intercambios que se realizan para la ordenación de la siguiente lista



**Paso a paso.**

**Pasada 1:** Momento inicial. Hay 10 elementos. Se divide la lista en 5 grupos.

8 43 17 6 40 16 18 97 11 7

Se comparan 8 y 16. No hay intercambio.

Se comparan 43 y 18. Se intercambian **(Intercambio 1).**

8 18 17 6 40 16 43 97 11 7

Se comparan 17 y 97. No hay intercambio.

Se comparan 6 y 11. No hay intercambio.

Se comparan 40 y 7. Se intercambian **(Intercambio 2).**

8 18 17 6 7 16 43 97 11 40

Fin Pasada 1.

**Pasada 2:** La distancia anterior era de 5. Como su división entera es 2, ahora la distancia de los intercambios será 2. Se divide la lista en 2 grupos.

8 **18** 17 **6** 7 **16** 43 **97** 11 **40**

Se comparan 8 y 17. No hay intercambio.

Se comparan 17 y 7. Se intercambian **(Intercambio 3).**

8 **18** 7 **6** 17 **16** 43 **97** 11 **40**

Se comparan 8 y 7. Se intercambian **(Intercambio 4).**

7 **18** 8 **6** 17 **16** 43 **97** 11 **40**

Se comparan 17 y 43. No hay intercambio.

Se comparan 43 y 11. Se intercambian **(Intercambio 5).**

7 **18** 8 **6** 17 **16** 11 **97** 43 **40**

Se comparan 17 y 11. Se intercambian **(Intercambio 6).**

7 **18** 8 **6** 11 **16** 17 **97** 43 **40**

Se comparan 8 y 11. No hay intercambio.

Se comparan 18 y 6. Se intercambian **(Intercambio 7).**

7 **6** 8 **18** 11 **16** 17 **97** 43 **40**

Se comparan 18 y 16. Se intercambian **(Intercambio 8).**

7 **6** 8 **16** 11 **18** 17 **97** 43 **40**

Se comparan 6 y 16. No hay intercambio.

Se comparan 18 y 97. No hay intercambio.

Se comparan 97 y 40. Se intercambian **(Intercambio 9).**

7 **6** 8 **16** 11 **18** 17 **40** 43 **97**

Se comparan 18 y 40. No hay intercambio.

Fin Pasada 2.

**Pasada 3:** La brecha de 2 pasa a ser de 1. Se ordena la lista restante.

7 6 8 16 11 18 17 40 43 97

Se comparan 7 y 6. Se intercambian **(Intercambio 10).**

6 7 8 16 11 18 17 40 43 97

Se comparan 7 y 8. No hay intercambio.

Se comparan 8 y 16. No hay intercambio.

Se comparan 16 y 11. Se intercambian **(Intercambio 11).**

6 7 8 11 16 18 17 40 43 97

Se comparan 8 y 11. No hay intercambio.

Se comparan 16 y 18. No hay intercambio.

Se comparan 18 y 17. Se intercambian **(Intercambio 12).**

6 7 8 11 16 17 18 40 43 97

Se comparan 16 y 17. No hay intercambio.

Se comparan 18 y 40. No hay intercambio.

Se comparan 40 y 43. No hay intercambio.

Se comparan 43 y 97. No hay intercambio.

Fin Pasada 3.

Así, concluimos que el ShellSort realizará un total de **3 pasadas y 12 intercambios.**

1. (10) Dada una lista voto[0.......n-1], donde cada elemento de lista representa un voto en las elecciones. Suponga que cada voto se da como un número entero que representa el ID del candidato elegido. Desarrolle un algoritmo para determinar quién gana la elección. Determine la complejidad del algoritmo

**R:** Revisar el archivo 5\_votos.py.

El algoritmo en cuestión es la función votación() (línea 39).

El algoritmo no tiene en cuenta el caso de empate. En tal caso, quien tenga un número de ID menor será considerado ganador.

El algoritmo utiliza las funciones mergeSort() de la línea 3 y eliminar\_duplicado\_específico() de la línea 29.

Sabemos que la función mergeSort() tiene una complejidad de

La función eliminar\_duplicado\_específico() solo utiliza un ciclo while y no realiza operaciones de complejidad mayor a , por lo que su complejidad es de

Finalmente, la función votación() utiliza un ciclo for en el que se encuentra la función eliminar\_duplicado\_específico(). Sin embargo, este for no depende del número de votos, sino del número de candidatos. Así, en el caso de que cada candidato haya obtenido un solo voto, habría que recorrer el ciclo n veces, pero la complejidad eliminar\_duplicado\_específico() pasa a ser de , al no haber valores repetidos. Así mismo, en caso de que solo un candidato se haya llevado todos los votos, el ciclo for se recorrerá una sola vez, pero eliminar\_duplicado\_específico() mantiene su complejidad. En un caso promedio, para m candidatos, el ciclo se recorre m veces, mientras que la función eliminar\_duplicado\_específico(), realizará un aproximado de n/m pasos. Multiplicando, obtenemos que la complejidad del for termina siendo de **.**

Así, realizando la suma de las complejidades de las funciones involucradas, obtenemos que la complejidad total del algoritmo será de , ya que corresponde a la función de mayor complejidad que es mergeSort().

1. (10) Se cuenta con una lista de tuplas

futbolistasTup = [(1, "Casillas"), (15, "Ramos"), (3, "Pique"), (5, "Puyol"), (11, "Capdevila"), (14, "Xabi Alonso"), (16, "Busquets"), (8, "Xavi Hernandez"), (18, "Pedrito"), (6, "Iniesta"), (7, "Villa")]

si se aplica futbolistasTup.sort(key=lambda futbolista: futbolista[0])

1. ¿Qué resultado se obtiene al aplicar el método .sort()?

**R:** Se ordena la lista de tuplas de manera ascendente por el número de cada futbolista.

1. ¿Qué se está especificando en los parámetro (key=lambda futbolista: futbolista[0])?

**R:** Se está especificando que las tuplas se ordenen según su primer elemento que, en este caso, consiste en el número del futbolista. De esta manera, al momento de ordenar las tuplas, solo se tiene en cuenta el primer elemento de cada tupla.

1. Aplique este método a las listas de los puntos 1,3, 4. Que conclusión puede obtener.

**R:** El método .sort() de las listas es útil para realizar ordenamientos de elementos de la misma clase en las listas, de manera que no es necesario escribir las funciones de ordenamientos, sin embargo, para calcular la complejidad, necesitaríamos saber que algoritmo utiliza este método para ordenar los datos, de manera que sepamos que efectivamente es eficiente.

1. Por favor según opinión realice una tupla con los mejores inventos del 2019

donde usted califica el que más le gusta o le parece importante.

**Anotación** la escala con la que usted cuenta es de 1 a 100 (no tiene que asignar ninguno de los extremos si no lo desea).

**R:** inventosTup = [(18, 'Analogue Mega SG'), (20, 'Osso VR'), (22, 'KaiOs'), (34, 'IDAGO'), (45, 'Dyson Airwrap'), (60, 'WeWALK'), (71, 'LighSail 2'), (75, 'ECONcrete'), (78, 'AeroFrams'), (82, 'Airthings Wave Plus'), (84, 'PathSpot')]

1. (10) Diseñe e implemente una función para encontrar todos los valores negativos dentro de una lista dada. Tu función debería devolver una nueva lista que contiene los valores negativos.

**R:**  Revisar 7\_negativos.py

¿Cuándo ocurre el peor de los casos y cuál es el tiempo de ejecución para ese caso?

El peor de los casos ocurre cuando todos los números de la lista son negativos. El algoritmo recorre toda la lista independientemente de si tenga o no números negativos. La diferencia se encuentra en que en caso de que haya números negativos, se debe realizar la operación adicional de agregar los números a la nueva lista.

Tenemos:

-En la línea 4 se realiza una asignación **(1 OE)**

-En la línea 5 se entra a un ciclo que realiza una comparación y un incremento. **(2 OE)**

-En la línea 6 se entra a un condicional que entra a la lista y realiza una comparación **(2 OE).**

-En la línea 7 se entra a dos listas **(2 OE)**.

-En la línea 8 se hace un retorno **(1 OE).**

Ahora, en el peor de los casos tenemos:

-Se realiza la asignación **(+1)**

-Se entra al ciclo. Se realiza una asignación y n incrementos. Se realizan n + 1 comparaciones **(+2n + 2)**

-Se entra al condicional n veces **(+2n)**

-Se añade un elemento a la nueva lista n veces **(+2n)**

-Se realiza el retorno **(+1)**

Así:

Así, el tiempo de ejecución será proporcional al número de datos, teniendo que hacer un total de 6n + 4 pasos para completar el algoritmo en el peor caso. Así podemos decir que tiene una eficiencia de orden

1. (5) Dada la siguiente lista de números:

[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

¿Cuál sería el resultado a la lista que después de 3 llamadas recursivas de ordenamiento por mezcla?

**R:** En la tercera llamada a la función, la lista tendría el siguiente orden:

[1, 21, 26, 45, 28, 29, 2, 9, 16, 49, 27, 39, 34, 43, 40, 46]

¿Por qué?

**R:** En la primera llamada la lista se divide en 2:

[21, 1, 26, 45, 29, 28, 2, 9] [16, 49, 39, 27, 43, 34, 46, 40]

Para retornar, hace una segunda llamada y divide las listas en 2:

[21, 1, 26, 45] [29, 28, 2, 9] [16, 49, 39, 27] [43, 34, 46, 40]

Para retornar hace la tercera llamada y divide las listas en 2 nuevamente:

[21, 1] [26, 45] [29, 28] [2, 9] [16, 49] [39, 27] [43, 34] [46, 40]

La cuarta llamada deja los elementos unitarios, y deja las listas como se encuentran en la tercera llamada. Luego, en esta se ordenan los datos de cada sublista, así:

[1, 21] [26, 45] [28, 29] [2, 9] [16, 49] [27, 39] [34, 43] [40, 46]

Es decir, que la lista estaría en la tercera llamada así:

[1, 21, 26, 45, 28, 29, 2, 9, 16, 49, 27, 39, 34, 43, 40, 46]

Que era lo que habíamos planteado.

1. (5)Dado el siguiente código evalué su complejidad.

Determine que hace este esté algoritmo

Explique cada uno de sus funciones y sus clases



El código corresponde a la implementación de una clase Conjunto, de manera que se puedan utilizar en otros contextos. No se encuentra un programa principal, por lo que podríamos decir que el código por sí solo no hace nada. Ahora, si vamos a revisar cada método, podemos definir la complejidad de cada uno:

La función init() corresponde al constructor de la clase Set. Crea un clase lista y la asigna al atributo elementos del objeto creado. Así, solo realiza dos operaciones y su complejidad es de **O(1).**

La función len() retorna el tamaño del conjunto, accediendo al tamaño de la lista Elementos. Así, solo realiza dos operaciones y su complejidad es de **O(1).**

La función findPosition() encuentra la posición de un elemento si el conjunto está ordenado. Si el elemento no se encuentra, retorna 0. Utiliza la búsqueda binaria en la lista Elementos, por lo que su complejidad es de **O(log(n))**

La función contains() recibe un elemento a buscar y retorna un booleano que puede ser True si el elemento se encuentra en la lista Elementos y False en otro caso. Como accede a la función findPosition(), podemos decir que su complejidad es de **O(log(n))**

La función add() recibe un elemento a agregar, determina si se ya se encuentra en el conjunto, en cuyo caso, no lo agrega y en caso contrario, lo agrega. Como accede a la función findPosition(), podemos decir que su complejidad es de **O(log(n))**

La función remove() recibe un elemento a eliminar del conjunto, determina si se encuentra en el conjunto, en cuyo caso, lo elimina y en caso contrario envía un mensaje. Como accede a la función findPosition(), podemos decir que su complejidad es de **O(log(n))**

La función isSubsetOf() recibe un Conjunto y retorna un booleano que puede ser True en caso de que todos los elementos del conjunto se encuentran en el conjunto parámetro, y False en caso contrario. Utiliza un ciclo for que recorre todos los elementos del conjunto, por lo que su complejidad es **O(n).**

Finalmente, la función iter(), define un objeto iterador para el conjunto, de manera que se pueda recorrer el conjunto. Hace falta esa fracción de código, sin embargo, como corresponde a la construcción de un objeto, podemos deducir que tiene complejidad **O(1).**

1. (10) Construya un algoritmo para encontrar un valor específico en una matriz de valores ordenada por filas y columna.

El algoritmo toma como entrada una matriz de valores donde cada fila y cada columna están en orden, junto con un valor para ubicar en esa matriz. Devuelve si ese elemento existe en la matriz.

Por ejemplo, dado la siguiente matriz y buscar el 7, el algoritmo daría como resultado **sí**

Pero si se pide encontrar el número 0, el algoritmo daría como resultado **no**



**R:** Revisar 10\_busqueda\_matriz.py. Específicamente, el método buscar\_elemento. Se me ocurrió tarde que podía utilizar LinkedLists para mejorar el proceso. Ya lo haré en otra ocasión.

1. (10) se tiene una lista A con 100 elementos A[ a1……a100 ]

B de 60 elementos B[ b1……b60 ]

Se desean resolver las siguientes tareas

1. Ordenar cada lista aplicando el método Quicksort
2. Crear una lista C que sea la unión de la lista A y B
3. Ordenar la lista C y visualizarla

**R:** Revisar 11\_listas\_quick.py

1. (5) Investigue en que consisten las técnicas de ordenación Radixsort y Binsort, en lo posible haga un ejemplo

**R:** Los códigos los encontré en la página Geeks for Geeks.

La técnica de ordenación Radix consiste en realizar el ordenamiento de las listas por unidades, luego por centenas y así sucesivamente hasta que se alcance la cifra más significativa (la mayor). El algoritmo consiste en ubicar en una nueva lista de 10 índices cada elemento según la cifra que se esté evaluando y crear una lista nueva con los valores en orden. Por ejemplo (sacado de Wikipedia):

Tenemos [25, 57, 48, 37, 12, 92, 86, 33].

Creamos una lista de índices y ubicamos en cada uno, los números cuya cifra de unidades corresponda con el índice:

0:

1:

2:12 92

3:33

4:

5:25

6:86

7:57 37

8:48

9:

Y se crea una lista con cada uno de estos elementos en el nuevo orden, así:

[12, 92, 33, 25, 86, 57, 37, 48]

Se repite el proceso, esta vez teniendo en cuenta el número de las decenas:

0:

1:12

2:25

3:33 37

4:48

5:57

6:

7:

8:86

9:92

Y al final obtendremos la lista ordenada: [12, 25, 33, 37, 48, 57, 86, 92]

Ahora la técnica de ordenación por casilleros (BinSort o BucketSort) es un algoritmo que filtra los números de la lista en casilleros. Cada casillero va a exigirle al número determinada condición, normalmente un rango. Dentro de cada casillero entonces se realiza un ordenamiento individual con otro algoritmo de ordenación y al final se unen todos los casilleros en una lista ordenada.

Utilicemos la lista anterior con algunos elementos más:

Tenemos [25, 22, 57, 21, 48, 37, 43, 12, 45, 63, 92, 86, 33].

Vamos a utilizar 10 casilleros con un rango de 10 cada uno (es decir, de 0 a 9, de 10 a 19, etc.), y en cada uno vamos a ubicar los números que corresponden al rango:

0:

1: 12

2: 25, 22, 21

3: 37, 33

4: 48, 43, 45

5: 57

6: 63

7:

8: 86

9: 92

Y ordenamos cada casillero por otro algoritmo de ordenación:

0:

1: 12

2: 21, 22, 25

3: 33, 37

4: 43, 45, 48

5: 57

6: 63

7:

8: 86

9: 92

Y concatenamos todo al final en una lista ordenada:

[12, 21, 22, 25, 33, 37, 43, 45, 48, 57, 63, 86, 92]

En casa me tomaré el tiempo para comprender la implementación en Python. Gracias.