**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**

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]

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

¿Cuál es la eficiencia del método?

Compárela con la eficiencia del punto 1

* Puesto a que el algoritmo que planteo es una doble búsqueda lineal por el for que recorre cada elemento y por el uso del método count de notación O(n), termina en que su complejidad Big O es O(n^2), y por la naturaleza de la búsqueda hace igual el tiempo de ejecución de cualquier tipo de lista que se le pase. Por lo tanto, la eficiencia del punto 1 es igual a la del punto 2 en mi caso.

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 asi

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

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

Justifique su respuesta

* Se está usando ordenamiento por inserción ya que para la segunda pasada (o sea cuando esté parado en el 32) ya todo está ordenado, mientras que para el burbuja le cuesta una pasada más, al igual que al de selección.

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



* Pasadas en total = 22
* Pasadas donde se realizan cambios = 4

1) En la primera pasada donde se realizan cambios se intercambia el 18 con el 11 y el 40 con el 11.

2) En la segunda se intercambia el 7 con el 97, luego el 16 con el 7, luego el 6 con el 7 y finalmente el 7 con el 43.

3) En la tercera se intercambia el 16 con el 6.

4) En la cuarta se intercambia el 43 con el 11, luego el 11 con el 17, luego el 17 con el 43 y luego el 16 con el 43, luego el 40 con el 43, luego se colocó el 6 al principio de la lista y finalmente el 40 con el 18, y el 43 con el 40.

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

De la forma en que implementé mi algoritmo tiene una complejidad O(n).

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. Que resultado se obtiene al aplicar el método .sort

* Ordena la lista según el número del jugador de menor a mayor.

1. Que se esta especificando en los parámetro (key=lambda futbolista: futbolista[0])

* Se le dice al sort que organize los elementos de la posición 0 de las tuplas.

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

* El método resulta muy bueno para ordenar listas. Es bastante rápido.
* Sort() está basado en Tim Sort, el cual comparando tiempos de ejecución vs Shell Sort, resulta más rápido (aunque por poco). 0.0004 en promedio para Shell Sort y 0.0003 en promedio para Tim Sort.

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

Donde usted califica el que mas 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)

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.

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

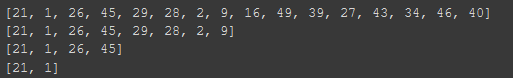
* El peor caso será O(n) ya que está basado en una búsqueda linear.

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?

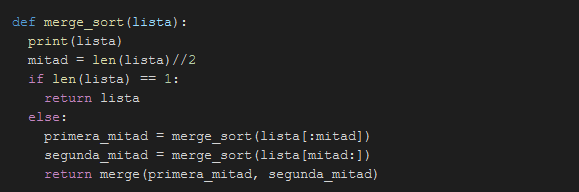
¿Por qué?



Segundo llamado

Tercer llamado

Primer llamado



Acá se hace el llamado recursivo que divide las listas.

* Esto sucede ya que estamos dividiendo las listas hasta que estas sean de 1 único elemento. Esto implica que para la siguiente pasada vamos a ver por pantalla [21]

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

Determine que hace este esté algoritmo

Esplique cada uno de sus funciones y sus clases



* **Complejidad:** O(n)
* **Clase Set**: Esta clase es una abstracción de un conjunto de datos ordenados.
* **Sus funciones:**

1. **\_\_init\_\_(**self**):** Inicializa un nuevo conjunto vacío.
2. **\_\_len\_\_(**self**):** Encuentra el tamaño del conjunto.
3. **\_\_contains\_\_(**self, element**):** Verifica si un elemento hace parte del conjunto.
4. **add(**self, element**)**: Añade un nuevo elemento al conjunto si este no existía antes.
5. **remove(**self, element**):** Elimina un elemento del conjunto si este ya existía antes.
6. **isSubsetOf(**self, setB**):** Verifica que un conjunto sea subconjunto de otro.
7. **\_\_iter\_\_(**self**):** Itera sobre los elementos de la lista.
8. **\_findPostion(**self, element**):** Encontra el índice del elemento ingresado.
9. (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**



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 un lista C que sea la unión de la lista A y B
3. Ordenar la lista C y visualizarla
4. (5) Investigue en que consisten las técnicas de ordenación Radixsort y Binsort, en lo posible haga un ejemplo
5. **Counting Sort:** Para poder hablar del radix sort, debemos saber qué es el counting sort. Este método de ordenamiento hace uso del ordenamiento de enteros, el cual ordena datos por sus llaves enteras. Además, permite hace muchos más tipos de operaciones debido a que estaremos usando una llave entera, en lugar de posiciones de la lista que solo nos permite hacer comparaciones entre ellas.

Este algoritmo se basa en contar las repeticiones de un elemento en una lista A y guardarlo en otra lista C a la cual cada índice corresponde al valor que se está contando y en esa posición va a almacenar esas repeticiones.

Tomemos como ejemplo A = [0,0,0,1,3,2,1,4,4,5] luego su C sería C = [3,2,1,1,2,1]

Fíjese que para C[0] está 3, el cual es exactamente la cantidad de repeticiones del elemento ‘0’ en A, luego en C[1] está el número de repeticiones de ‘1’ en A y así sucesivamente.

Teniendo esa lista, podríamos pensar que para ordenar la lista podríamos decirle que en una lista B fuera colocando los elementos de la forma en que los tenemos en C, o sea, que coloque primero 3 ceros, 2 unos, 1 dos…

***0,0,0,1,1,2,3,4,4,5***

Y en efecto, podemos hacer eso, aunque no de una manera tan explícita. Debemos crear un C modificado donde iremos contando el número de elementos que están detrás de la última repetición del elemento incluyéndose así mismo.

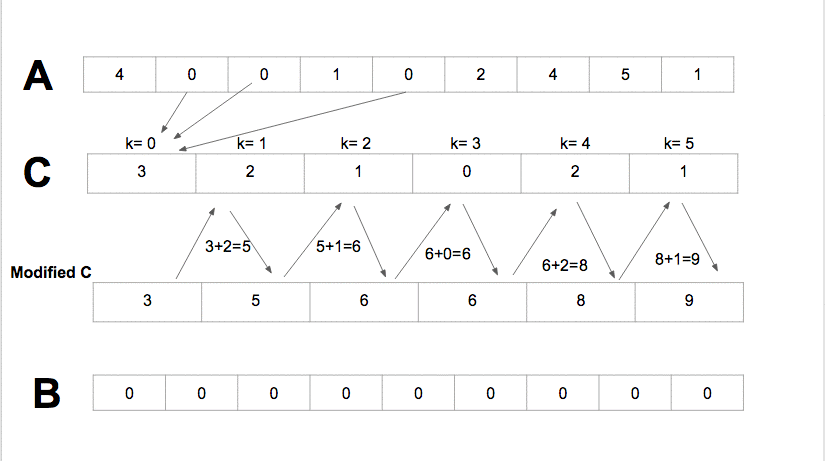
En este caso se vería de la siguiente manera:

C\_modicado = [3,5,6,7,9,10]

Ahora, cada elemento de A representa un índice en C modificado (por ejemplo a 5 en A le corresponde el índice 5 en C modificado), luego al recorrer A del último elemento hacia atrás, iremos accediendo a la posición que le corresponde a cada elemento de A en C modificado y le iremos restando uno. Esto nos dará el índice a dónde ese elemento de A debe parar.

Como vamos del último hacia atrás en A, entonces nos paramos en 5 el cuál tiene su índice en C modificado. Luego accedemos a C\_modificado[5] siendo este 10 y le restamos 1, esto nos da 9. 9 será el índice que le corresponde a 5 en la nueva lista organizada, luego pasamos a 4 en A, aplicando lo mismo, nos da que su nuevo índice será 8 y cuando lleguemos al otro 4 su índice será 7. Así sucederá hasta que lleguemos al inicio de la lista de A.

Se ilustra mejor con la siguiente imagen.



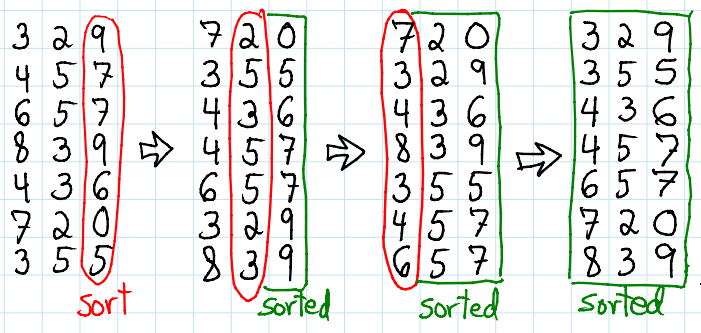
Como podemos observar, este método es muy eficiente y dinámico gracias a esta forma bastante inteligente de organizar los datos. Su complejidad es de O(n) a comparación del Ω(nlogn) para los algoritmos basados en comparaciones.

1. **Radix Sort:** Radix sort es un algoritmo de ordenamiento basado en el Counting Sort. Radix sort, separa una lista por columnas y va ordenando de los valores menos significantes a los más significantes basado en un sistema numérico *b* establecido (radix), tal que cada número posee *d* dígitos tal que d = |logb(k)+1|. Por ejemplo, en un *b* de 10, este algoritmo ordenaría por counting sort primero los que están en la posición 10^0, luego los que están en la posición 10^1 y así sucesivamente.

Radix sort es un ordenamiento estable, lo que significa que mantiene el orden relativo de los elementos que tienen el mismo valor clave. Por ejemplo, sea B una lista tal que B = [51, 42, 53, 54, 60] y sea *b*(el radix) = 10.

Para la primera ‘pasada’ del algoritmo explorará los elementos que pertenezcan a kx10^0, los cuales en este caso serían las unidades de cada número en la lista (1,2,3,4,0), ordenándolos desde que (0<1<2<3<4). Luego en la segunda cuando explore los elemento de kx10^1 (5,4,6) ordenándolos desde que (4<5<6). Se dará cuenta de que existen varios 5, lo que hará que revise sus relativos (1<3<4) y así mantendrá el orden relativo de las decenas a las unidades de cada elemento de la lista para no generar incongruencias. Finalmente, a la tercera pasada, tomando en cuenta lo anterior, tendremos ya la lista ordenada completamente: [42,51,53,54,60].

Ejemplo gráfico:



1. **Bin Sort:** Este algoritmo se basa dividir los elementos en grupos llamados canastas, definidas por un rango numérico (ej. 0-10, 10-20, …) en donde se organizan los elementos que contiene cada canasta con un algoritmo diferente o por medio de un llamado recursivo al bin sort. Luego, se reúnen de nuevo y forman la lista organizada al iterar sobre cada canasta e ir posicionando cada elemento en esta nueva lista. Personalmente me recuerda mucho al merge sort.

Por ejemplo, sea A = [0, 18, 2, 17, 20, 9]

* Creamos una lista canasta de n-tamaño que contenga los rangos que deseamos usar. En este caso, usaré 0-10, 10-20. Entonces canasta[0] contendrá los elementos que estén en el rango 0-10 y canasta[1] los que estén entre 10-20.
* Ponemos cada elemento de la lista en su respectiva canasta.
* Organizamos cada canasta. En este caso podría usar el método sort() para hacerlo.
* Iteramos sobre la canasta[0], colocamos sus elementos en una lista nueva que contendrá los elementos ya organizados. Se vería así: [0,2,9]
* Hacemos lo mismo con canasta[1], agregándole sus elementos a la misma lista de antes.
* Nos queda [0,2,9,17,18,20]