

Juan Felipe Martínez Bedoya

Taller número 6

Los puntos que se ven vacíos es por que su implementación es por código y se encuentra en adjunto en Python en un programa que se llama Codigos.py

1) Repetidos sin ordenar (repetidos1)

2) Repetidos ordenados (repetidos2).

Para comparar su complejidad se va a basar por tiempo y se va a encontrar la notación BigO. Para el tiempo se corrieron ambos programas 1000 veces, y se sacó un promedio de cuanto se demoró en correr ambos algoritmos usando la misma lista ordenada [4,7,11,4,9,5,11,7,3,5]. Realizando dicho procedimiento se encontró que la función repetidos1 toma 5.250453948974609e-06 segundos y la función repetidos2 toma 2.727985382080078e-06 segundos, por lo que es casi un 50% más rápido. En caso de la notación Big O se tiene que para el caso del método repetidos1 es $O(n^2 + n + 2)$ y para el método repetidos2 es $O(2n + 2)$. Como se puede observar en la notación Big O, el primer método tiene un orden cuadrático, y el segundo es lineal, por lo que siempre será mucho mejor el algoritmo de repetidos2.

3) Se trata de un algoritmo de ordenamiento por inserción, ya que el resto de los elementos se encuentra en la posición que quedarían si se mueven a la derecha y se van insertando al principio de las listas los elementos menores.

4)

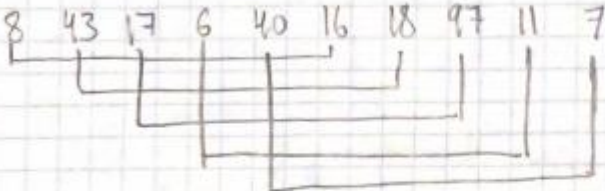
Lista original:

8 43 17 6 40 16 18 97 11 7

Elementos = 10

Pasada 1:

Salto de 5

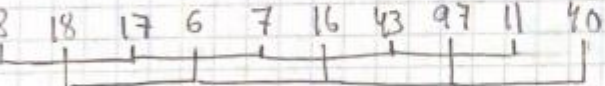


Nueva lista:

8 18 17 6 7 16 43 97 11 40

Pasada 2:

Salto de 2

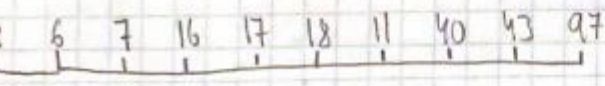


Nueva lista:

8 6 7 16 17 18 11 40 43 97

Pasada 3:

Salto de 1:



Nueva lista:

6 7 8 11 16 17 18 40 43 97

5) Algoritmo contador de votos (votos)

Para este caso se realizó lo mismo que el punto 2 para obtener un promedio de la efectividad por tiempo de la función fotos utilizando la lista [2,2,2,2,2,3,3,3,3,4,4,5,5,5,5,6,2,2,4,5,6,2,3,2,2,2,2,2,3,4,3,5], donde se

obtuvo un tiempo de 1.1081695556640624e-05 segundos. Ya la notación BigO es $O(n^2 + 5n + 4)$, como se puede ver esta es de orden cuadrático y bastante ineficiente.

6) Tuplas.

- a) Al aplicar el método `sort()` sin ningún parámetro nos devuelve la lista ordenada por el primer parámetro que se encuentre.
- b) Al intentar utilizar el método `futbolistasTup.sort(key=lambda futbolista : futbolista[0])` al fin y al cabo esta instrucción lo que hace es decidir el parámetro por lo cual se desea organizar, es algo que se le va a aplicar a cada elemento de la lista y basarse de eso para el ordenamiento. Ya que, si por ejemplo cambio los parámetros a `futbolistasTup.sort(key=lambda futbolista : futbolista[1])`, me ordena la lista dependiendo del nombre de los deportistas y no por el número de su camiseta.
- c) Para las listas del punto 1,3 y 4 no se puede utilizar el `key`, ya que estas no son listas con ningún nivel de complejidad, solo almacena números enteros simples, por lo que si aplicamos esto nos tira un `'TypeError'` ya que los objetos `int` no son contenedores de nada.
- d) Para esto busqué inventos del 2019 del siguiente link: (https://www.elconfidencial.com/alma-corazon-vida/2019-12-02/100-mejores-inventos-2019-time-558_2360991/) y tomé los que me parecieron más interesantes y la calificación que les doy es por utilidad que les encuentro:

Inventos = [(“ OrCam MyEye2”, 95), (“Osso VR”,75), (“Bose Frames”, 60), (“ECONcrete”, 90), (“Roybi Robot”, 80), (“LighSail 2”, 90), (“Brain Robitics AI”, 98), (“Postmates Serve”, 70), (“ Minecraft Earth”, 40), (“Theranica Nerivio”, 65), (“B'zt”, 85), (“ Genny”, 96), (“Draper / Sprout”, 97), (“AIR-.INK”, 55)]

7) Algoritmo buscador de números negativos (negativos)

El peor de los casos se da cuando todos los elementos de la lista son negativos, y para hallar un tiempo de ejecución promedio se utilizó la siguiente lista de 10 elementos `[-1,-1,-1,-1,-1,-1,-5,-6,-8,-8]` y se calculó un tiempo promedio de ejecución como se ha hecho en los puntos anteriores, este es de `4.986763000488281e-06` segundos. Adicionalmente se sacó su notación Big O, que es $O(2n + 2)$.

- 8) En la tercera llamada iterativa del método de ordenamiento por mezcla se seguiría dividiendo la lista en 2, por el momento se tendrían las siguientes sublistas: `[21, 1]` `[26, 45]`. Esto se da a que primero se dividen la lista de la izquierda y después la de la derecha, en este caso vemos que todavía se tienen está dividiendo el principio de la lista (lo más a la izquierda), y esto se seguirá dando hasta que la longitud de estas listas sea de 1, y cuando todo sea de esta longitud se comienza a unir la lista ordenándola.
- 9) Se tiene una clase llamada Set, cuyo constructor (`__init__`) lo que hace es crear una lista vacía, que es el único atributo que tiene esta clase. Empezando a explicar los métodos que se encuentran en esta clase se tienen: un método `__len__`, que sirve para calcular la longitud de la lista; un método `__contains__`, que sirve para ver si un elemento se encuentra o no en la lista; `add`, que sirve para adicionar nuevos elementos a la lista; `remove`, para remover un elemento que se encuentre en la lista de ella misma; `isSubsetOf`, sirve para determinar si la lista atributo se encuentra en su totalidad en otra lista aparte; `__iter__`, este método empieza a enviar elementos de la lista cada vez que es llamado, empezando desde el primero y entregando continuamente hasta que se acabe la lista; y finalmente se tiene el método `_findPosition`, este lo que hace es realizar una búsqueda binaria para encontrar un elemento. La complejidad de cada método es:

Función	Notación BigO
<code>__init__</code>	$O(1)$
<code>__len__</code>	$O(1)$
<code>__contains__</code>	$O(n+2)$
<code>add</code>	$O(n + \log n + 1)$
<code>remove</code>	$O(n + \log n + 1)$
<code>isSubsetOf</code>	$O(nk + n + 1)$
<code>__iter__</code>	$O(1)$
<code>_findPosition</code>	$O(\log n)$

10) Algoritmo para buscar un elemento en una matriz (busq_matriz).

11) Listas A y B.

- a) Para ordenar las listas por el método de Quick sort, se puede crear una función que cumpla el algoritmo planteado, o se puede utilizar el método predeterminado de Python `.sort()` o `sorted()` que se basa en dicho algoritmo para ordenar listas. Sería de esta manera:

`A.sort()` – `sorted(A)` – `quick_sort(A)`

`B.sort()` – `sorted(B)` – `quick_sort(B)`

- b) Para juntar las dos listas se pueden hacer dos cosas, o se usa el operador '+' o se usa el método `.extend()`. Sería de la siguiente manera:

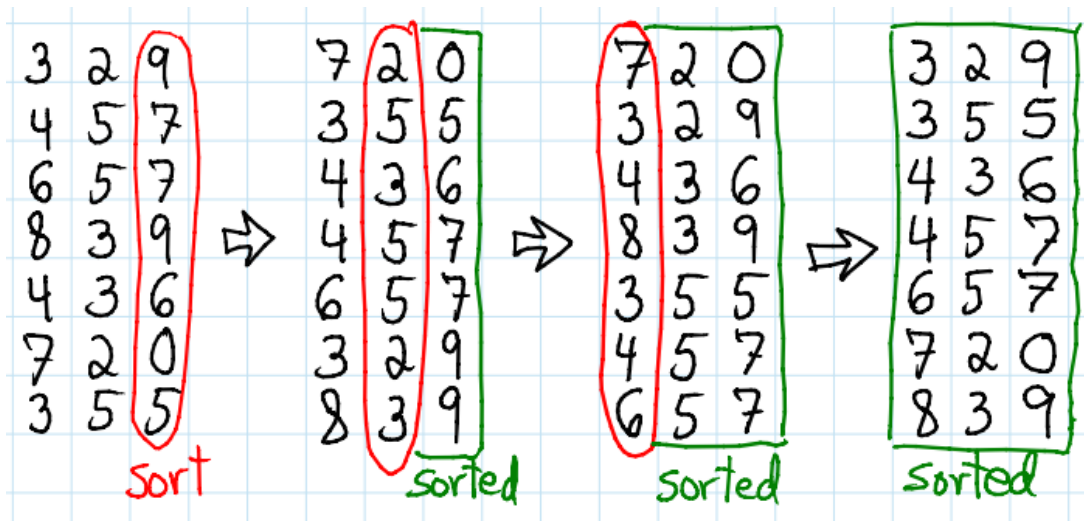
`C = A + B` – `C = A.extend(B)`

- c) Se ordena con cualquier método mencionado anteriormente y luego se imprime con un `print`. Sería de la siguiente manera:

`print(sorted(C))`

12) Radixsort: este tipo de ordenamiento sirve para listas simplemente enlazadas y numéricas, y se basa en la comparación de sus dígitos, empezando desde las unidades y se sigue hacia adelante (decenas, centenas, etc). Lo que se hace es que se coge la lista y se compara el dígito y se agrega a una lista aparte que contiene los elementos con el dígito (una lista de los que tengan el dígito 0, otra de 1, etc), luego se cogen los elementos de la lista aparte y se agregan en orden (empezando por los de 0 y terminando con los de 9), y así se hace con todos los

dígitos hasta terminar con todas las posiciones. (ver vídeo <https://www.youtube.com/watch?v=kA53tn0FoxU>)



Binsort: consiste en crear diferentes canastas (o bins en inglés, de ahí viene el nombre), donde se insertan los elementos cumpliendo la regla $\#bin = n(\text{número de elementos}) * lista[i]$ elemento en la posición i (u otra regla definido por el programador). Ya después cuando se llenan los canastos se aplica insertion sort en ellos y finalmente se agrega a la lista en orden (desde el primer canasto hasta el último. (ver vídeo <https://www.youtube.com/watch?v=VuXbEb5ywrU>)

BinSort example

- $K=5$. $list=(5,1,3,4,3,2,1,1,5,4,5)$

↓

Bins in array	
key = 1	1,1,1
key = 2	2
key = 3	3,3
key = 4	4,4
key = 5	5,5,5



Sorted list:
1,1,1,2,3,3,4,4,5,5,5