**Taller 2**

**Gian Paul Sánchez y Maria Paula Ayala**

**Cabe resaltar que los tiempos de ejecución van a ser distintos en todos los casos debido a que la rapidez del computador al reservar unos u otros espacios de memoria es diferente en todos los casos, aún así con la complejidad algorítmica se puede plantear cuál algoritmo es más rápido que otro. En conclusión al hacer este taller no confíamos en los tiempos de ejecución debido a la aleatoriedad de estos.**

**Punto 1 y 2:**

Como el algoritmo es el mismo en ambos casos, los dos tienen una complejidad asintótica igual a O(n). A su vez, ambos tienen un tiempo de ejecución similar por esta misma razón, ya que es el mismo código pero con la diferencia de ejecutarse en una lista distinta.

Tiempo de ejecución punto 1:

**Punto 3:**

El método de ordenación que se usó pudo ser inserción o selección. Con inserción en la primera pasada se tomaría un subarreglo de las primeras 2 posiciones, en este caso siendo 47 y 3, se comparan estos dos números y como el 3 es menor que el 47 se intercambian. En la segunda pasada el subarreglo aumenta una posición y quedan 3, 47 y 21, el 21 que es la nueva adición se compara con los otros 2 valores y se acomoda en su posición, quedando el arreglo 3, 21 y 47 con los otros números del arreglo detrás sin arreglar, quedando: 3, 21, 47, 32, 56 y 92.

Por otra parte, con el método de selección también quedaría de la misma manera ya que se busca el menor número en el arreglo y se intercambia con el primer número del arreglo que no esté arreglado, es decir, en este caso el arreglo inicial estaba así: 47, 3, 21, 32, 56, 92. Entonces en la primera pasada el número “seleccionado” es el 47 y se busca cuál es el menor entre todo el arreglo, en este caso encontramos que es el 3, por lo cual se van a intercambiar el 47 y el 3, quedando el arreglo así: 3, 47, 21, 32, 56, 92. Y en la segunda pasada se “selecciona” el 47 también ya que coincidencialmente quedó de segundo, y se vuelve a buscar en todo el arreglo, sin contar ya el 3, cuál es el menor, encontramos que es el 21, entonces se intercambian el 21 con el 47, quedando el arreglo así: 3, 21, 47, 32, 56, 92.

**Punto 4:**

Comenzó de está forma: [8, 43, 17, 6, 40, 16, 18, 97, 11, 7]

Y todos estos fueron los cambios que se fueron dando durante el proceso. Cada Pasada corresponde al arreglo momentos antes de variar la distancia de comparación (gap) entre los elementos del arreglo, en otras palabras, corresponde al arreglo cada 1 iteración del ciclo más externo. Por otro lado, cada intercambio representa el cambio del arreglo cada vez que se comparan los elementos según este gap.

Intercambio # 1 -> [8, 43, 17, 6, 40, 16, 18, 97, 11, 7]

Intercambio # 2 -> [8, 18, 17, 6, 40, 16, 43, 97, 11, 7]

Intercambio # 3 -> [8, 18, 17, 6, 40, 16, 43, 97, 11, 7]

Intercambio # 4 -> [8, 18, 17, 6, 40, 16, 43, 97, 11, 7]

Intercambio # 5 -> [8, 18, 17, 6, 7, 16, 43, 97, 11, 40]

Pasada # 1 -> [8, 18, 17, 6, 7, 16, 43, 97, 11, 40]

Intercambio # 1 -> [8, 18, 17, 6, 7, 16, 43, 97, 11, 40]

Intercambio # 2 -> [8, 6, 17, 18, 7, 16, 43, 97, 11, 40]

Intercambio # 3 -> [7, 6, 8, 18, 17, 16, 43, 97, 11, 40]

Intercambio # 4 -> [7, 6, 8, 16, 17, 18, 43, 97, 11, 40]

Intercambio # 5 -> [7, 6, 8, 16, 17, 18, 43, 97, 11, 40]

Intercambio # 6 -> [7, 6, 8, 16, 17, 18, 43, 97, 11, 40]

Intercambio # 7 -> [7, 6, 8, 16, 11, 18, 17, 97, 43, 40]

Intercambio # 8 -> [7, 6, 8, 16, 11, 18, 17, 40, 43, 97]

Pasada # 2 -> [7, 6, 8, 16, 11, 18, 17, 40, 43, 97]

Intercambio # 1 -> [6, 7, 8, 16, 11, 18, 17, 40, 43, 97]

Intercambio # 2 -> [6, 7, 8, 16, 11, 18, 17, 40, 43, 97]

Intercambio # 3 -> [6, 7, 8, 16, 11, 18, 17, 40, 43, 97]

Intercambio # 4 -> [6, 7, 8, 11, 16, 18, 17, 40, 43, 97]

Intercambio # 5 -> [6, 7, 8, 11, 16, 18, 17, 40, 43, 97]

Intercambio # 6 -> [6, 7, 8, 11, 16, 17, 18, 40, 43, 97]

Intercambio # 7 -> [6, 7, 8, 11, 16, 17, 18, 40, 43, 97]

Intercambio # 8 -> [6, 7, 8, 11, 16, 17, 18, 40, 43, 97]

Intercambio # 9 -> [6, 7, 8, 11, 16, 17, 18, 40, 43, 97]

Pasada # 3 -> [6, 7, 8, 11, 16, 17, 18, 40, 43, 97]

Final -> [6, 7, 8, 11, 16, 17, 18, 40, 43, 97]

**Punto 5:**

El algoritmo tiene una complejidad de O(n).

**Punto 6:**

a) La lista de tuplas se ordena según el número del jugador de menor a mayor.

b) Lo que se hace con toda esta expresión es especificarle al método sort que queremos ordenar según el primer elemento de la tupla, que en este caso es el número del jugador. Por ejemplo, si hiciéramos futbolista: futbolista[1] se ordenaría según los nombres, entonces quedarían en orden alfabético.

c) Si aplicamos el método sort() a estos puntos lo que vamos a lograr es ahorrarnos el tiempo de crear un método de ordenación utilizando el que ya viene en python internamente.

d) inventos = [(“Bastón para ciegos”, 60),(“CirugíasVR”, 80),(“Airpods Pro”,55),(“Pantalla 8k”, 45),(“Prótesis de mano”,90),(“Domicilios automáticos”, 75),(“Robot ayudante”, 77)]

**Punto 8:**

La lista al comienzo es: [21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]

Al primer llamado recursivo la lista quedaría: [21, 1, 26, 45, 29, 28, 2, 9]

En el segundo llamado recursivo la lista a ordenar es: [21, 1, 26, 45]

Al tercer llamado recursivo la lista es: [21, 1]

Todo esto tomando en cuenta que los primeros llamados recursivos se hacen con el arreglo por la izquierda.

**Punto 9:**

El código implementa la idea de los sets mediante el uso de una lista, es decir crea listas con las mismas características que un set.

***- \_\_Init\_\_:*** Crea una lista vacía llamada *theElements*, esta funciona como nuestro “set”. Su complejidad es: O(1)

***- \_\_len\_\_:*** Retorna el número de elementos de la lista. Sucomplejidad es: O(1)

- ***\_\_contains\_\_:*** Lo que se hace dentro de este método es llamar al método *findPosition* para saber si el elemento que entra como parámetro está o no en la lista, luego se hacen unas verificaciones para saber con certeza si el índice entregado por *findPosition* es correcto y si no lo es se retorna False como diciendo que el elemento no se encuentra en la lista, por el contrario si retorna True significa que el índice retornado por *findPosition* es correcto y el elemento si se encuentra en la lista. Su complejidad es: O(log n)

- ***add:*** Revisa si el elemento a añadir no se encuentra en la lista *theElements*, si esto es cierto, entonces llama al método *\_findPosition* para encontrar la posición del elemento en la lista ordenada y después poder añadir en la misma posición este elemento en la lista *theElements*. Su complejidad es: O(log n)

- ***remove:*** Con el método de Python assert revisamos primero si el elemento que se quiere eliminar si está en la lista *theElements*, si no está sale el mensaje “The element must be in the set”. Sí sí se encuentra en la lista entonces se usa el método *\_findPosition* para hallar la posición exacta y luego se llama al método pop que sirve para eliminar de la lista la posición escogida. Su complejidad es: O(log n)

- ***isSubsetOf:*** Lo que hace este método es iterar elemento por elemento para saber si cada el elemento del set A está en el set B. Si hay algún elemento del set A que no está en el set B esto quiere decir que el primero no es subconjunto del segundo, devolviendo False. Por el contrario, si el ciclo se completa con éxito esto quiere decir que todos los elementos del set A están en el set B, dando como resultado True. Su complejidad es: O(n).

- ***\_\_iter\_\_:*** En este método se llama a otro el cual no está especificado en el código, pero podemos ver que su fin es recorrer la lista y devolverla. La complejidad de este método depende de la complejidad del método \_SetIterator así que no la podríamos especificar:

- ***\_findPostition:*** Es un método de búsqueda binaria el cual si se encuentra el elemento devuelve su posición, que será la variable de *mid* y si no lo encuentra devuelve la posición de la variable *low*. Su complejidad es: O(log n).

La complejidad de todo el código sería O(n) por ser el peor caso.

**Punto 12:**

*Radixsort:* Este ordenamiento sólo se puede para números, ya que se basa en “tomar” cada valor posicional que tenga el mayor número del arreglo en todos los números. Por ejemplo, si el mayor número es 1503, se revisará primero las unidades de cada elemento, luego las décimas, luego las centenas y por último la unidad de mil. Lo que se hace con cada valor posicional es que se acomoda en la posición del arreglo según su dígito, es decir, siguiendo con el ejemplo de 1503 cuando se revisen las unidades ira a la 3 posición, luego cuando se revisen las unidades irá a la posición 0, después en las centenas a 5 posición y por último a la 1 posición. Aunque hay que tener en cuenta que todos los números que tengan el mismo dígito se acomodaran en la misma posición y en orden en el que aparecieron en el arreglo.

Veamos un ejemplo: Se tiene el arreglo [ 170, 45, 75, 90, 802, 24, 2, 66] primero revisaremos las unidades [ 170, 45, 75, 90, 802, 24, 2, 66] y se ordenarán en su respectivas posiciones: [ 170, 90, 802, 2, 24, 45, 75, 66], como hay números repetidos se pone primero el que se encuentra primero en el arreglo. Seguimos ahora con las décimas [ 170, 90, 802, 02, 24, 45, 75, 66] y queda así: [ 802, 02, 24, 45, 66, 170, 75, 90]. Por último, las centenas [ 802, 002, 024, 045, 066, 170, 075, 090] y el arreglo va a quedar así: [ 2, 24, 45, 66, 75, 90, 170, 802]. Ya se acaban las iteraciones y el arreglo efectivamente quedó ordenado.

*Binsort:* El ordenamiento bin sort, o también conocido como bucket sort, trata de crear unas “canecas” que son un tipo de arreglo, o también pueden ser listas enlazadas, de las cuales en cada una irán unos “intervalos” de números. Es decir, si en el arreglo el número máximo es 25 y el menor es el 1, se pueden crear 5 “canecas” las cuales se repartirían así: la caneca #1 va desde el 1 hasta el 5, la #2 desde el 6 hasta el 10, la #3 desde el 11 hasta el 15, la #4 desde el 16 hasta el 20 y la #5 desde el 21 hasta el 25. Después de que todos los elementos del arreglo están en su respectiva caneca cada una se organiza con un método de ordenamiento, puede ser insertion sort. Por último, ya se devuelven los elementos de las canecas al arreglo, se van ubicando en orden de las canecas, es decir primero se pone en el arreglo toda la caneca #1, luego la #2, y así hasta llegar a la #5.

Veamos un ejemplo: Tenemos el arreglo [11, 9, 21, 8, 17, 14, 13, 1, 24, 12, 16, 19, 22], y crearemos las canecas así:

#1, que va del 1 al 5

#2, que va del 6 al 10

#3, que va del 11 al 15

#4, que va del 16 al 20

#5, que va del 21 al 25.

Al repartir los elementos del arreglo en las canecas quedarían así:

#1 -> 1

#2 -> 9, 8

#3 -> 11, 14, 13, 12

#4 -> 17, 16, 19

#5 -> 21, 24, 22

Después se organiza cada caneca, y queda así:

#1 -> 1

#2 -> 8, 9

#3 -> 11, 12, 13, 14

#4 -> 16, 17, 19

#5 -> 21, 22, 24

Por último ya cada caneca vuelve al arreglo en orden, es decir primero la #1, luego la #2, y así sucesivamente, por lo que el arreglo ya quedaría ordenado, así: [ 1, 8, 9, 11, 12, 13, 14, 16, 17, 19, 21, 22, 24]