Trabajo Práctico: Algoritmos de búsqueda y ordenamiento

Estructuras de datos
Comisión A (Turno Mañana)
Segundo cuatrimestre de 2020

Docente:	Ariel Clocchiatti
Fecha de entrega:	21 / 09 / 2020
Alumnos:	Colquhoun, Noelia Kener, Sebastian

Índice

1. Introducción

En este trabajo se abordarán problemas de búsqueda y ordenamiento de los elementos en un arreglo. Para ello se trabajará con la integración de los siguientes temas:

- Programación en Python.
- Algoritmos recursivos.
- Arreglos.

1.1. Algoritmos de búsqueda y ordenamiento

Los algoritmos de búsqueda y ordenamiento son ampliamente utilizados en la resolución de problemas informáticos. En la mayoría de los casos, trabajar con datos ordenados hace que los algoritmos sean más eficientes. La búsqueda es uno de los problemas que más se beneficia al trabajar con datos ordenados. Por lo cual, estos algoritmos están muy relacionados entre sí.

Para simplificar se utilizaran arreglos unidimensionales y que contengan sólo números enteros (sin embargo es posible usar cualquier estructura de datos). También se supondrá que el algoritmo completo se puede resolver en memoria, osea que los arreglos serán de un tamaño reducido, y los ordenamientos de tipo numérico y de menor a mayor.

1.2. Eficiencia y complejidad

Cuanto menor sea la complejidad computacional, los algoritmos son más eficientes y por consiguiente, más rápidos. La complejidad depende de la cantidad de datos que tenemos que procesar. Cuando tenemos un algoritmo para resolver un problema, tenemos que medir los recursos computacionales que el algoritmo va a necesitar:

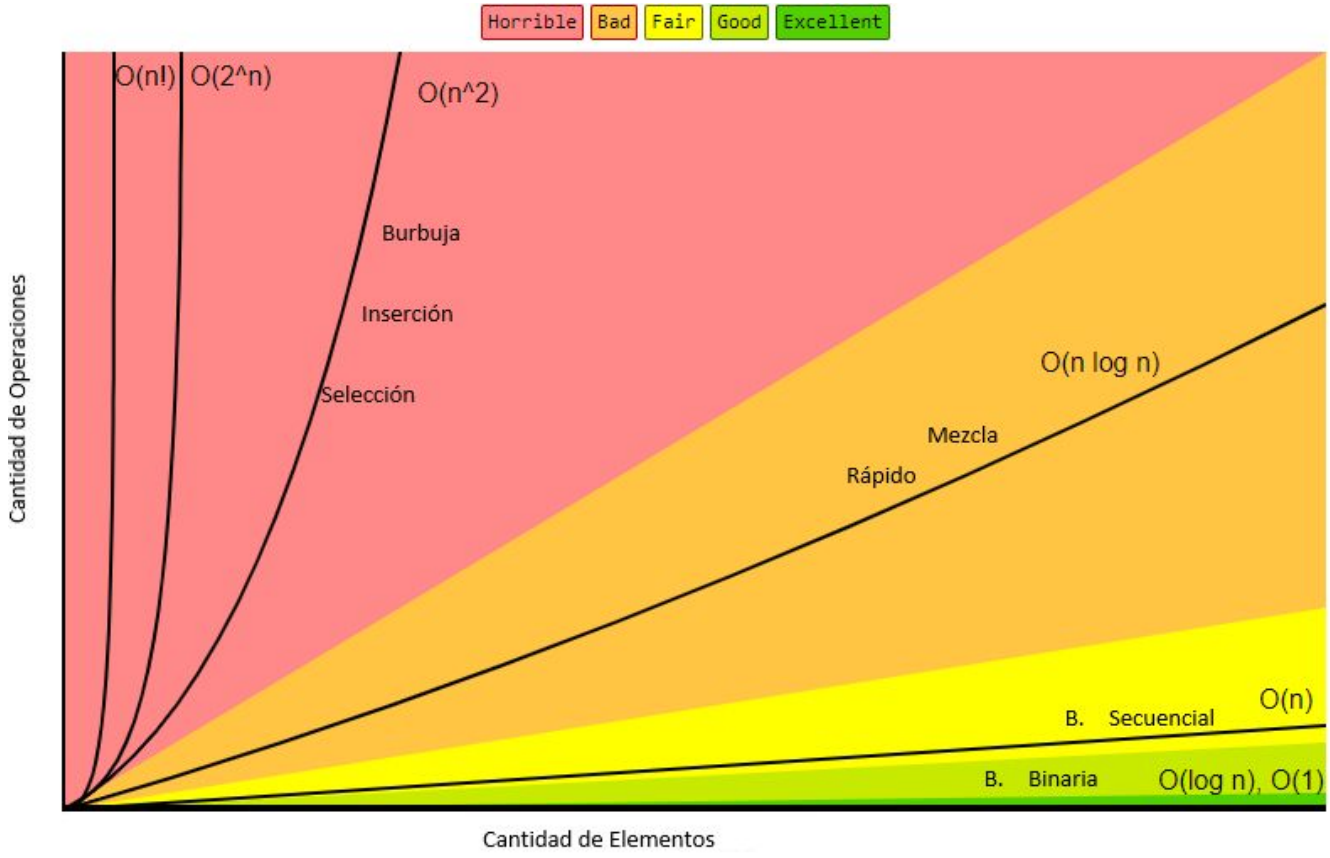
- Tiempo de ejecución en función de la cantidad de datos (cantidad de operaciones que realiza el algoritmo).
- Cantidad de memoria en función de la cantidad de datos.

Estas dos medidas conforman la complejidad computacional del algoritmo. La complejidad espacial se reduce al resolverlos en memoria. Así que se verá la idea de **complejidad temporal**.

La complejidad temporal mide la cantidad de operaciones básicas que realiza el algoritmo (sumas, restas, multiplicaciones, divisiones, operadores de comparación, operadores lógicos, etc), y se representa usando una notación que quiere decir "del orden de..."(O(...)).

- O(log2(n)): Complejidad logarítmica, es decir, la cantidad de operaciones que hace el algoritmo está en el orden del log2 de la cantidad de datos.
- O(n): Complejidad de orden lineal, es decir, la cantidad de operaciones que hace el algoritmo está en el orden de la cantidad de datos.
- O(nlog2(n)): Complejidad lineal por logarítmica.
- O(n2): Complejidad cuadrática.
- O(2n): Complejidad exponencial.

Complejidades en función de la cantidad de datos: a menor complejidad, más eficiencia.



Para estudiar estos algoritmos se verán los casos promedios, ya que de esa manera veremos la funcionalidad más cercana a la realidad posible.

2. Objetivos

- Comprender los conceptos de algoritmos de búsqueda y ordenamiento.
- Investigar y estudiar el funcionamiento de los algoritmos.
- Implementar los algoritmos en Python.
- Comparar la eficiencia de los distintos algoritmos según su complejidad computacional.

3. Informe

3.1. Conceptos

Búsqueda

Es el proceso en el cual se puede, o no, encontrar un ítem particular dentro de un conjunto de ítems. Una búsqueda normalmente devuelve "true" o "false", sin embargo, en ocasiones se modifica el código para que devuelva una dirección. También se puede combinar con ordenamiento, cumpliendo ambas funciones en un mismo algoritmo.

Ordenamiento

Sort (en inglés) es el proceso de modificar o disponer de datos en algún orden secuencial específico. Por ejemplo codigo de articulo o alfabeticamente.Se puede dar de forma ascendente o descendente, su propósito fundamental es facilitar el proceso de búsqueda, acortar tiempo.

3.2. Tipos de algoritmos

En este trabajo estudiaremos e implementaremos el algoritmo de **búsqueda binaria** y el algoritmo de **ordenamiento de burbuja**. Serán comparados con los algoritmos restantes.

Búsqueda

- Búsqueda lineal o secuencial (Linear search).
- Búsqueda binaria (Binary search).

Ordenamiento

- Ordenamiento por inserción (Insertion sort).
- Ordenamiento por selección (Selection sort).
- Ordenamiento de burbuja (Bubble sort).
- Ordenamiento por mezcla (Merge sort).
- Ordenamiento rápido (Quicksort).

3.3. Ordenamiento: Método Burbuja (Bubble sort)

Este método, a grandes rasgos, consiste en que los valores más pequeños de la lista "burbujean" quedándose en las posiciones superiores, mientras que los valores mayores quedan más abajo. Hace una comparación de elementos contiguos, en cada pasada el elemento mayor queda al final y el más chico retrocede una posición, por lo tanto en cada pasada hace menos comparaciones que en la anterior.

Supongamos el siguiente vector: A = [50, 20, 40, 80, 30].

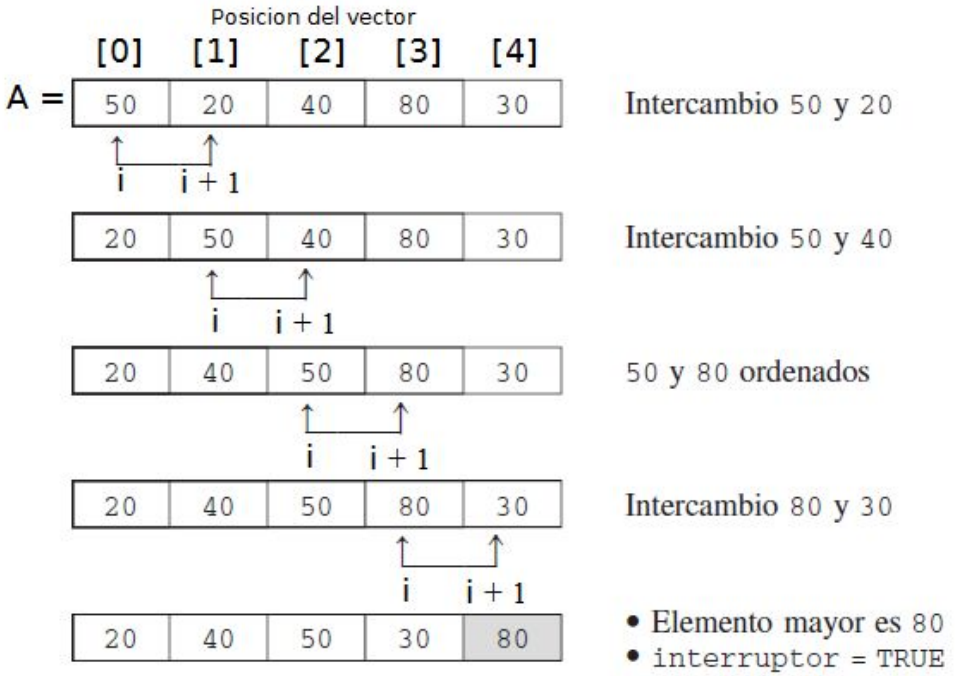


Figura2: Pasada 0 en el vector A. El interruptor no es más que una bandera, indica si se produjo un intercambio

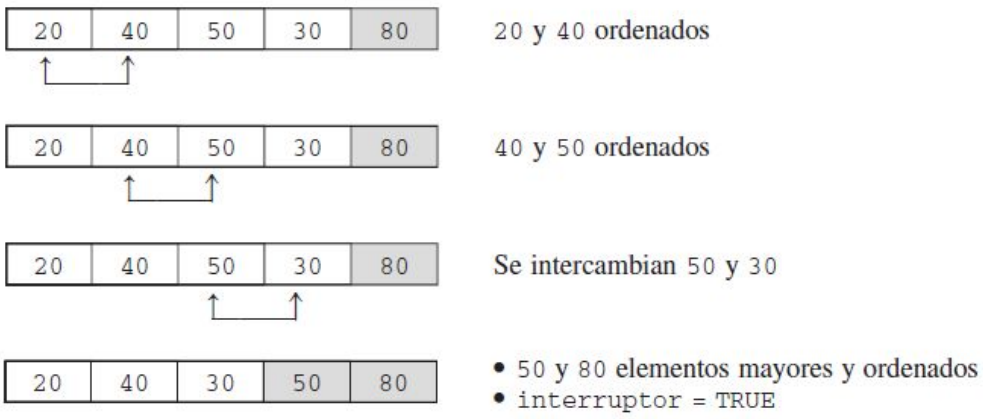


Figura 3: Pasada 1 en el vector A.

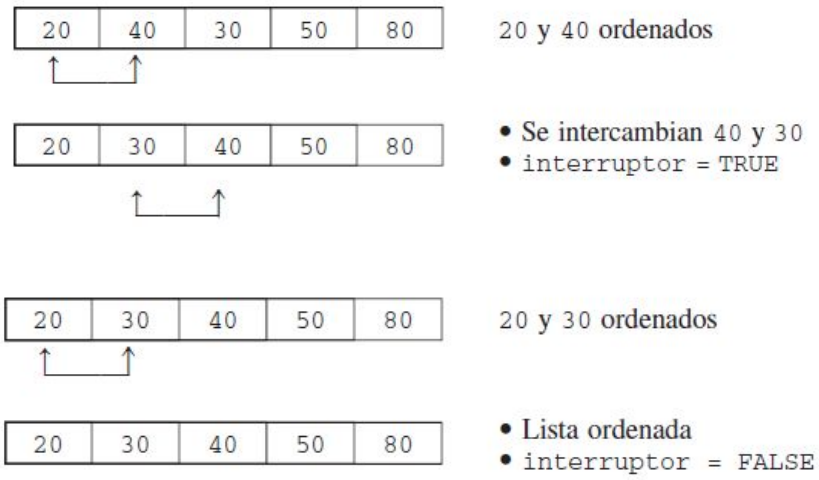


Figura 4: Pasada 2 en el vector A.

Figura 5: Pasada 3 en el vector A.

3.4. Implementación

****Adjunta en archivo de Google Colab para su análisis**

3.5 Comparación de eficiencia: Método burbuja frente a los otros métodos

	Burbuja	Rápido	Mezcla	Selección	Inserción
Descripción **E= elemento**	Compara un E con el siguiente y si este es menor, los intercambia. Va del 1° al último E. Al llegar al último, si aún no están ordenados hace pasadas sucesivas hasta que esto suceda.	Elige un E del arreglo (pivot). Luego acomoda todos los elementos menores a la izquierda del mismo y los mayores a la derecha. De esta forma ya está ubicado correctamente el pivot. luego se ordenan los otros elementos generando 2 sublistas. el proceso se repite para cada sublista.	Divide el arreglo en sub arreglos y estos los vuelve a dividir si puede. Luego los arreglos se ordenan y se mezclan de a 2 ordenando a la vez los elementos de cada subarreglo. Cuando se mezclen los últimos 2 arreglos estará ordenado el vector.	Busca el E con menor valor del arreglo y lo coloca en la 1ra posición del vector. Luego busca el siguiente menor valor y lo coloca en la 2da posición. Así sucesivamente hasta que el arreglo queda completamente ordenado.	Selecciona un E y lo compara con el anterior. Si no hay E anterior, toma el E siguiente. Si el E seleccionado es menor al anterior, los intercambia y sigue comparando hasta llegar al 1er E. Luego vuelve al último y repite la secuencia.
Ventajas	*Sencillo *Código reducido *Eficaz	*Muy rápido *No requiere memoria adicional	*Estable para arreglos chicos	*Fácil implementación *No requiere memoria adic. *Pocos intercambios	*Fácil implementación *Requiere poca memoria
Desventajas	*Lento *Muchas lecturas/escrituras en memoria	*Implementación más complicada *Recursividad (usa muchos recursos)	*Definido recursivamente (no recursivo = memoria adicional)	*Lento y poco eficiente para arreglos grandes o medianos.	*Lento *Numerosas comparaciones
Complejidad Mejor caso	O(n^2)	O(n)	O(n log n)	O(n^2)	O(n)
Promedio	O(n^2)	O(n log n)	O(n log n)	O(n^2)	O(n^2)
Peor caso	O(n^2)	O(n^2)	O(n log n)	O(n^2)	O(n^2)

En cuanto a eficiencia en relación entre cantidad de datos y tiempo de ejecución, los métodos de **inserción** y **selección** son muy similares, ya que estos métodos en cada pasada harán un único movimiento. En la inserción se toma un elemento y se va agregando el resto en cada pasada, de manera ordenada, y en la selección en cada pasada se elige el menor y se acomoda decrecientemente en el arreglo. Por este motivo tanto uno como el otro dependen de la cantidad de datos: aumentarán su tiempo en vectores chicos y lo disminuirán con los grandes.

Un escalón arriba en cuanto a velocidad de ejecución están el ordenamiento por **mezcla** y el método **Quicksort**. Ya que al tratarse de métodos recursivos hace el mismo procedimiento en el mismo momento varias veces. La mezcla separa el array hasta la mínima expresión y hace el proceso inverso ordenando los elementos. El Quicksort hace un proceso similar, solo que, toma de un pivote(por lo general en el medio de la lista) y divide el arreglo en mitades, hasta llegar a su mínima expresión ordenando a un lado y otro del pivote por mayor y menor. La razón por la cual la velocidad aumenta es porque hace mucha más comparaciones en cada pasada, es decir analiza más mitades en cada vuelta. Por ejemplo:

vuelta 1 -> Análisis de una lista

vuelta 2 -> Análisis de dos listas

vuelta 3 -> Análisis de tres listas

etc.

Por su parte, el **método de burbuja** ofrece una muy alta facilidad de comprensión y programación pero, por el contrario, es uno de los **menos eficientes**. Dependiendo de su versión, puede o no tener variables como "interruptor" , la eficiencia en cuanto a tiempo de ejecución en relación a la cantidad de datos varía entre 0(n2) y 0(n).

En la versión más simple se hacen n − 1 pasadas y n − 1 comparaciones en cada pasada. Por consiguiente, el número de comparaciones es (n − 1) * (n − 1) = n2 − 2n + 1, es decir, hace tantas pasadas y tantas comparaciones como elementos en el vector.

En las versiones mejoradas, con la variable interruptor por ejemplo, la eficiencia será diferente en cada caso. En el mejor de los casos se hará una sola pasada (cuando la lista ya esté ordenada), y en el peor de los casos se harán todas las pasadas, comparaciones e intercambios.

En conclusión la ordenación por burbuja puede terminar en menos pasadas, pero requiere muchos más intercambios que la ordenación por selección y su promedio es mucho más lenta, sobre todo cuando los arreglos a ordenar son grandes.

3.6 Búsqueda: Método de búsqueda Binaria (Binary search)

La **búsqueda secuencial** se aplica a cualquier lista. En ella, luego de comparar el valor buscado con el 1er ítem del arreglo, si éste no coincide, hay a lo sumo n-1 ítems restantes para verificar si alguno es el valor que buscamos.
Si la lista está ordenada, la **búsqueda binaria** proporciona una técnica de búsqueda mejorada: esta comienza analizando el ítem central de nuestro arreglo/lista. Si el elemento coincide, la búsqueda termina. Si el elemento que buscamos es mayor que el que se encuentra en ese centro, ya podemos descartar la mitad inferior de la lista y el ítem central (y a la inversa si es menor)
Podemos entonces repetir el proceso con la mitad superior. Comenzar en el ítem central y compararlo con el valor que estamos buscando. Una vez más, o lo encontramos o dividimos la lista por la mitad, eliminando por tanto otra gran parte de nuestro espacio de búsqueda posible.

Después de dividir la lista suficientes veces, terminamos con una lista con un único elemento. O bien es el que buscamos, o no. De ambas formas la búsqueda termina.

3.7 Implementación

****Adjunta en archivo de Google Colab para su análisis**

3.8 Comparación de eficiencia: Método de búsqueda Binaria frente a la búsqueda secuencial

	Binaria	Secuencial
Descripción	Va dividiendo el arreglo a la mitad y comparando el elemento buscado con el valor del índice central.	Compara el valor de cada índice del vector con el elemento buscado.
Ventajas	*Reduce el tiempo de búsqueda de un elemento. *En listas grandes, en solo una comparación reduce el listado a la mitad.	*Fácil de implementar. *No requiere que la lista esté ordenada.
Desventajas	*Solo funciona si el arreglo está ordenado	*No es eficiente en arreglos largos
Orden de Complejidad	O(log 2 n)	O(n)
Mejor caso	O(1)	O(1)
Promedio	O(½ log 2 n)	O(n)
Peor caso	O(log 2 n)	O(n)

4. Bibliografía

- https://runestone.academy/runestone/static/pythoned/SortSearch/toctree.html
- https://www.bigocheatsheet.com/
- https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx