

Datos Generales

Título del trabajo: “Algoritmos de Búsqueda y Ordenamiento en Python”

Alumnos: (Cain Cabrera – caincito.cabrera7@gmail.com) (Enzo Chavez - enzoachavez@gmail.com)

Materia: Programación I

Fecha de Entrega: 09/06/2025

Introducción:

El presente trabajo busca desarrollar como en la programación, **los algoritmos de búsqueda y ordenamiento** son fundamentales para el manejo eficiente de la información, y cómo es que se logra localizar eficientemente los elementos que se necesiten dentro de un conjunto de datos.

Entender cómo usarlos, cuándo y en qué situaciones se aplican es clave para el desarrollo de software. Este trabajo explora estos algoritmos, da a conocer sus definiciones y luego pone en caso práctico algunos de ellos para evidenciar su utilidad.

Marco Teórico:

Los algoritmos de ordenamiento permiten re-ordenar un conjunto de datos teniendo en cuenta y respetando un criterio asignado así como por ejemplo de menor a mayor. Alguno de los más utilizados son:

Bubble Sort: compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Bueno para colecciones pequeñas ordenadas y su peor caso es cuando la lista está desordenada al revés de como debería ser.

Insertion Sort: construye una lista ordenada tomando elementos uno a uno e insertándolos en la posición adecuada. Bueno para listas pequeñas que estén ordenadas o casi ordenadas y su peor caso es cuando la lista está en orden inverso.

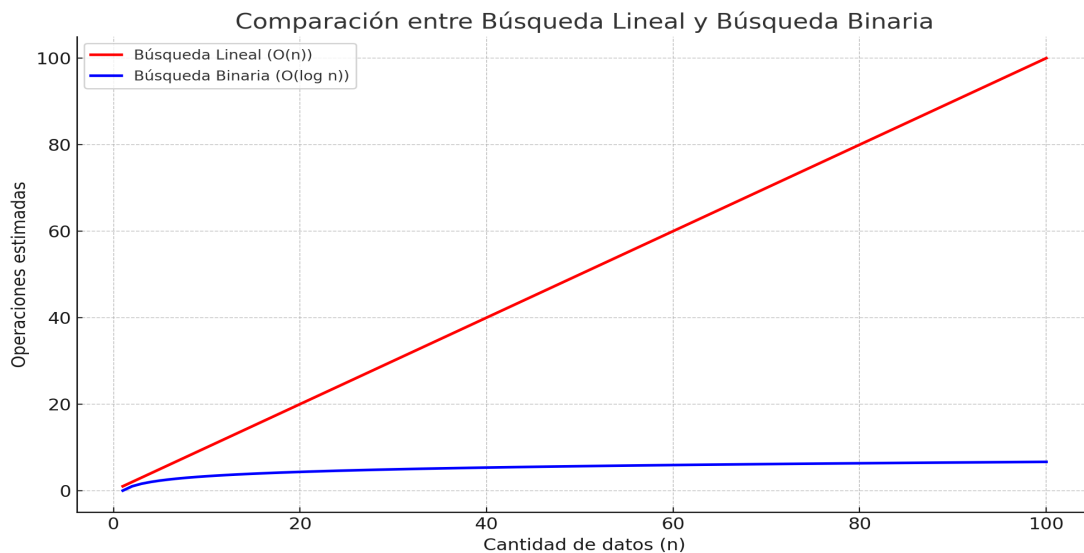
Selection Sort: Funciona seleccionando el elemento más pequeño en la lista de elementos y lo reubica a la primera posición, luego busca el siguiente más pequeño y lo pone en segunda posición, así sucesivamente hasta el final de la lista. Al comparar todos los números para saber cual es el mínimo, su peor o mejor caso da igual.

Quicksort: Selecciona un elemento de la lista como “pivote” y lo utiliza para dividir la lista en 2 sublistas, una con elementos menores y otra mayores. Luego funciona recursivamente aplicando este método a cada sublista. Muy bueno para listas grandes cuando el pivote las divide en partes iguales o casi iguales.

Por otro lado, **los algoritmos de búsqueda** tienen como objetivo encontrar uno o más elementos dentro de una colección. Dentro de estos encontramos:

Búsqueda lineal: recorre uno por uno todos los elementos de forma secuencial hasta encontrar el objetivo lo que lo vuelve simple pero ineficiente en colecciones largas debido al tiempo que le toma recorrerlo hasta el final por ejemplo.

Búsqueda binaria: nos exige que la colección esté ordenada y nos devuelve resultados mucho más rápido que la búsqueda lineal en conjuntos de datos grandes. Funciona dividiendo las colecciones en mitades sucesivas hasta encontrar el elemento deseado.



Comparación: Búsqueda lineal contra Búsqueda binaria

Criterio	Búsqueda Lineal	Búsqueda Binaria
Velocidad	$O(n)$	$O(\log n)$
Requisito previo	No necesita datos ordenados.	Necesita datos ordenados.
Uso de memoria	Solo la lista original.	Misma lista, pero puede requerir ordenamiento previo.

De esta informacion e investigacion nos surgieron algunas preguntas que consideramos importante desarrollar en este trabajo:

¿En todos los casos se debe utilizar el método de primero ordenar y luego buscar?

La respuesta a través de la investigación es que **no siempre es así, y que debe haber una adaptación** como en los siguientes casos:

Caso 1: Primero ordenar para que luego al buscar sea más rápido.

Un caso simple para entenderlo puede ser para encontrar un DNI de una lista grande de DNI's, al estar ordenados se simplifica muchísimo la búsqueda al poder utilizar la búsqueda binaria.

Caso 2: Primero se busca, y luego se ordena.

Esto pasa cuando no se busca específicamente un elemento, sino que buscamos coincidencias y luego, al encontrarlas, se ordena como por ejemplo por tiempo, relevancia, etc.

Tal es el caso del SEO en las páginas web, en las que primero se buscan todas las coincidencias y luego se las ordena para presentar los resultados al usuario.

Caso 3: Casos híbridos.

También existen casos más complejos que son híbridos, en los que los elementos se semi-ordenan para una búsqueda más eficaz pero al finalizar la misma los elementos son nuevamente reordenados para finalizar el proceso y hacer una devolución y presentación más ordenada y específica para el usuario.

Conclusión: Si bien en muchos casos ordenar los datos antes de buscar es una estrategia óptima para acelerar las búsquedas, en situaciones reales como bases de datos, motores de búsqueda y sistemas distribuidos, se emplean también formas donde primero se filtran o buscan los datos relevantes y luego se ordenan según distintos criterios.

Otra de las preguntas que nos surgieron fueron sobre el uso de la memoria de estos algoritmos y ¿Como se puede responder ante exigencias/entradas más grandes como sucede en empresas grandes?

Teniendo en cuenta que la búsqueda binaria requiere que los datos estén ordenados previamente, y que ese mismo ordenamiento puede usar más memoria (dependiendo del algoritmo que se use para ordenar), encontramos que existe la escalabilidad de la memoria de forma horizontal y vertical que son formas de mejorar los recursos o añadir más para poder seguir siendo eficientes y soportar la carga:

Escalabilidad vertical: Consiste en **aumentar los recursos del mismo nodo o máquina:** más RAM, mejor CPU, mejor disco.

Escalabilidad horizontal: Consiste en **agregar más máquinas/nodos al sistema,** creando una red.

Conclusión: Elegir entre búsqueda lineal o binaria no depende solo del algoritmo, sino también de las condiciones de memoria, velocidad y estructura de datos. A medida que los volúmenes crecen, los sistemas escalan verticalmente (más recursos en una sola máquina) o horizontalmente (más máquinas). Estos tipos de escalabilidad buscan balancear velocidad, uso de memoria y responder a las exigencias de mayores cargas y necesidades del sistema.

Caso Práctico:

Se desarrolló un programa para ordenar y buscar dentro de un legajo de alumnos. El programa utiliza dos algoritmos de ordenamiento, selection sort y bubble sort, y dos de búsqueda, búsqueda binaria y lineal, dando así diferentes escenarios al programa. Los siguientes algoritmos son:

```
def busqueda_binaria(lista,objetivo):
```

```
    inicio = 0
```

```
    fin = len(lista) - 1
```

```
    while inicio <= fin:
```

```
        medio = (inicio + fin) // 2
```

```
        if lista[medio] == objetivo:
```

```
            return medio
```

```
        elif lista[medio] < objetivo:
```

```
            inicio = medio + 1
```

```
        else:
```

```
            fin = medio - 1
```

```
    return -1
```

```

def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]

```

Metodología utilizada:

- Buscamos información en vídeos, tanto de la facultad como de YouTube nutriendo cada aspecto del tema dado.
- Vimos los algoritmos visuales para entender cómo funciona a alto nivel.
- Lo llevamos al código e hicimos pruebas llevando al límite cada algoritmo.

Conclusiones:

Los algoritmos de búsqueda y ordenamiento son una base fundamental que todo programador y desarrollador debe dominar. Ya que la función que tienen es sumamente importante a la hora de ordenar y buscar elementos sea el que sea el

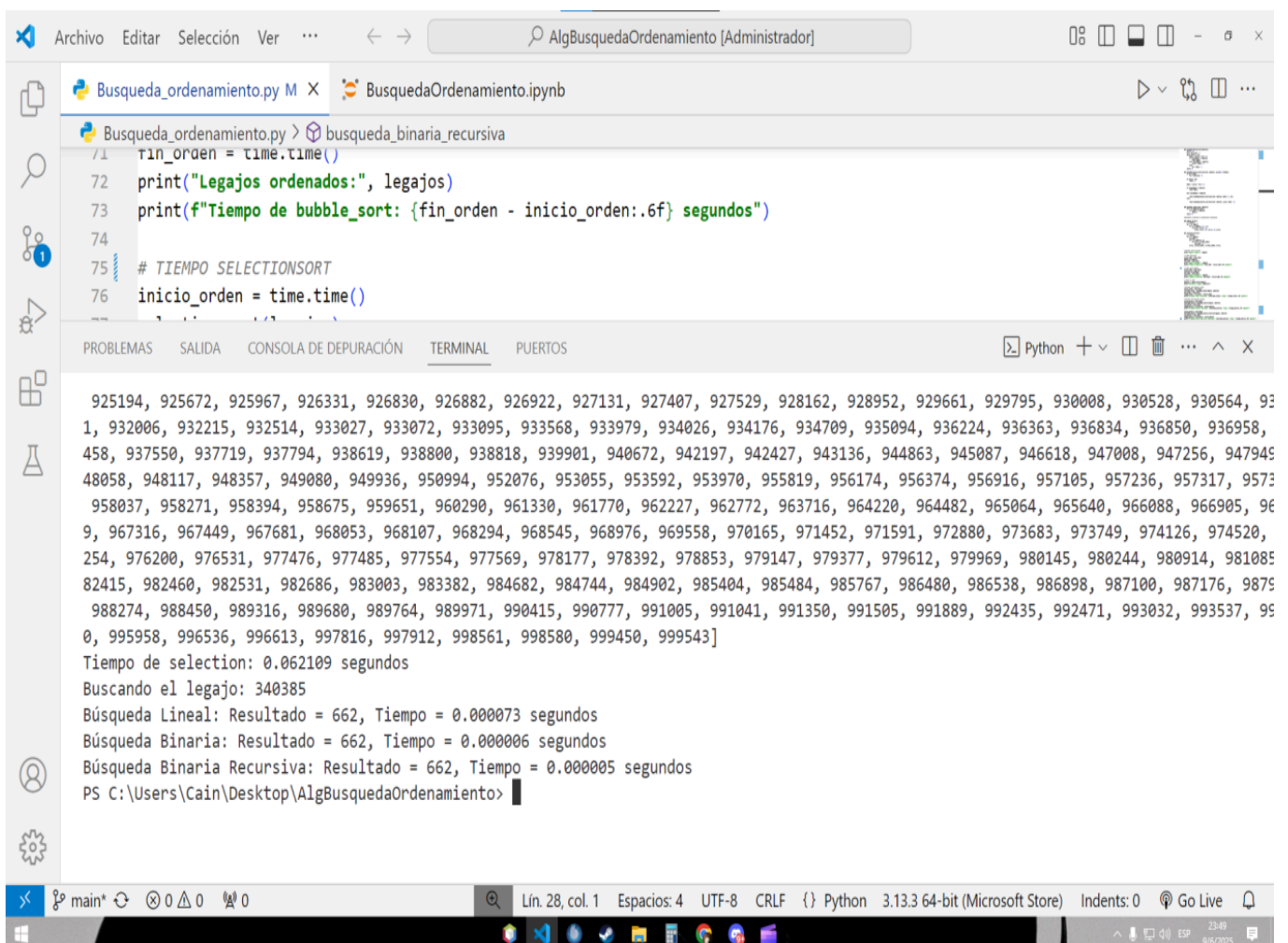
proyecto en el que estamos trabajando, entendiendo no solamente lo que hacen sino cómo afecta a bajo nivel nuestro rendimiento del programa.

La búsqueda binaria es uno de los algoritmos más eficiente siempre y cuando el conjunto de datos este ordenado, En caso de la búsqueda lineal, vemos que se aplica a conjuntos de datos desordenados, aunque su complejidad temporal no es eficiente, cumple su función de igual manera, siendo eficiente en conjunto de Datos pequeños. En conclusión, es importante antes de aplicar cualquier algoritmos, ver el contexto en el cual vamos a movernos, ya que así nos permite saber de antemano que algoritmos va a ser más eficiente para x caso, ya sea conocer la cantidad de elementos o el tipo de datos con el que vamos a trabajar.

Repositorio:

https://github.com/CainCabrera/Busqueda_Ordenamiento._Python/blob/main/Busqueda_ordenamiento.py

Foto del código funcionando:



The screenshot shows a Jupyter Notebook interface with a file named 'Busqueda_ordenamiento.py'. The code in the notebook is as follows:

```
1 fin_orden = time.time()
2 print("Legajos ordenados:", legajos)
3 print(f"Tiempo de bubble_sort: {fin_orden - inicio_orden:.6f} segundos")
4
5 # TIEMPO SELECTIONSORT
6 inicio_orden = time.time()
```

The output of the code is displayed in the terminal window at the bottom of the notebook:

```
925194, 925672, 925967, 926331, 926830, 926882, 926922, 927131, 927407, 927529, 928162, 928952, 929661, 929795, 930008, 930528, 930564, 931111, 932006, 932215, 932514, 933027, 933072, 933095, 933568, 933979, 934026, 934176, 934709, 935094, 936224, 936363, 936834, 936850, 936958, 937550, 937719, 937794, 938619, 938800, 938818, 939901, 940672, 942197, 942427, 943136, 944863, 945087, 946618, 947008, 947256, 947945, 948058, 948117, 948357, 949080, 949936, 950994, 952076, 953055, 953592, 953970, 955819, 956174, 956374, 956916, 957105, 957236, 957317, 957317, 958037, 958271, 958394, 958675, 959651, 960290, 961330, 961770, 962227, 962772, 963716, 964220, 964482, 965064, 965640, 966088, 966905, 967316, 967449, 967681, 968053, 968107, 968294, 968545, 968976, 969558, 970165, 971452, 971591, 972880, 973683, 973749, 974126, 974520, 976200, 976531, 977476, 977485, 977554, 977569, 978177, 978392, 978853, 979147, 979377, 979612, 979969, 980145, 980244, 980914, 981085, 982415, 982460, 982531, 982686, 983003, 983382, 984682, 984744, 984902, 985404, 985484, 985767, 986480, 986538, 986898, 987100, 987176, 987500, 988274, 988450, 989316, 989680, 989764, 989971, 990415, 990777, 991005, 991041, 991350, 991505, 991889, 992435, 992471, 993032, 993537, 993958, 995958, 996536, 996613, 997816, 997912, 998561, 998580, 999450, 999543]
Tiempo de selection: 0.062109 segundos
Buscando el legajo: 340385
Búsqueda Lineal: Resultado = 662, Tiempo = 0.000073 segundos
Búsqueda Binaria: Resultado = 662, Tiempo = 0.000006 segundos
Búsqueda Binaria Recursiva: Resultado = 662, Tiempo = 0.000005 segundos
PS C:\Users\Cain\Desktop\AlgBusquedaOrdenamiento>
```

The terminal output shows a large list of sorted numbers, the time taken for selection sort, and the results of three different search algorithms: Linear Search, Binary Search, and Recursive Binary Search. All three search algorithms found the element 340385 at index 662. The Binary Search and Recursive Binary Search algorithms are significantly faster than the Linear Search algorithm.