TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

UTN
UNIVERSIDAD TECNOLÓGICA NACIONAL

Trabajo Práctico Integrador

Búsqueda y ordenamiento

Alumnos:
- Nicolas Bressan – nicolasbressan00@gmail.com
- Franco Bertotti – francob1997@gmail.com

Materia: Programación I
Profesor coordinador: Alberto Cortez
Fecha de Entrega: 09 de junio de 2025

Índice

Introducción

Cuando estamos trabajando con grandes cantidades de datos, la eficiencia y el tiempo se vuelve muy importante para nuestros programas, por eso, la búsqueda y el ordenamiento se tornan una pieza fundamental para ayudarnos a trabajar grandes volúmenes de datos. Para lograr búsquedas eficientes y rápidas podemos recurrir a las binarias pero para ello debemos realizar ordenamientos anteriormente para lograr reducir los tiempos drásticamente. A continuación desarrollaremos cómo podemos lograr mejorar nuestros programas gracias a estos conceptos para hacer nuestros algoritmos más eficientes y rápidos.

Marco Teórico

Búsquedas

Tipos de búsquedas

Búsqueda lineal:

Es la búsqueda más simple y sencilla, donde se recorre uno a uno los elementos del vector o lista comparando con el objetivo para determinar si es igual, para realizar esta búsqueda no es necesario que los elementos estén ordenados ya que los comparará a todos. El tiempo de esta búsqueda se expresa de la siguiente forma: O(n), y será proporcional al tamaño de la lista.

Búsqueda binaria:

Esta búsqueda “parte” el vector o lista en 2, donde previamente los elementos deben ser ordenados, de menor a mayor, para que al dividirse se pregunte si el objetivo es menor o mayor que el elemento central que divide, así el algoritmo seleccionará qué mitad es donde seguirá buscando, se tirará de la misma forma hasta que solo se compare con el elemento buscado, y por ende sea o no el objetivo encontrado.

El tiempo en este tipo de búsqueda será expresado por O(log n) ya que es un logaritmo de n, y donde la diferencia de tiempo se vuelve significativamente menor a la búsqueda lineal, donde aquí al duplicar el tamaño de la lista se puede llegar a tardar casi el mismo tiempo, no así con la lineal. Para comprender mejor el funcionamiento vemos como lo expresó Cormen, Leiserson, Rivest y Stein.(2009), “It takes ,.n/ time towalkann-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child.” (p.288).

Búsqueda interpolar:

La búsqueda por interpolación es un método para encontrar un elemento en un conjunto de datos muy similar a la búsqueda binaria, y también la lista debe estar ordenada. La diferencia con la búsqueda binaria que siempre divide el vector o lista por la mitad, la búsqueda por interpolación es un poco más "inteligente": intenta adivinar o estima dónde podría estar el valor buscado basándose en la posición del valor buscado respecto a los valores que se encuentran en los extremos del rango de la búsqueda.

Su eficiencia en promedio es superior, pero puede volverse menos eficiente si los datos no están bien distribuidos, podríamos decir que si los datos tienen grandes saltos entre sí, la rapidez de búsqueda será menor a la binaria.

Búsqueda hash:

Es un tipo de búsqueda más avanzado para encontrar información de manera muy eficiente. Transforma directamente el dato que queremos buscar en una dirección o ubicación dentro de una “tabla hash", donde se almacenan los datos que querramos. Esto se logra a través de una operación llamada **función hash**.

La función hash nos indica rápidamente dónde estaría el elemento buscado. Esto hace que el tiempo de búsqueda sea prácticamente instantáneo, sin importar qué tan grande sea el conjunto de datos.

Cuando dos datos diferentes intentan ocupar la misma ubicación en la tabla, se conoce como "colisión" o “conflicto”. Para manejar estas situaciones, existen métodos específicos que aseguran que todos los datos puedan ser almacenados y encontrados correctamente, manteniendo la velocidad de la búsqueda.

Ordenamientos

Ordenamiento de listas:

Ordenar una lista desordenada según el criterio conveniente, algunos ejemplos que desarrollaremos como Bubble sort, Quick sort, Selection sort e Insertion sort.

Tipos de ordenamientos

Bubble sort

“Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.” (2025) https://www.geeksforgeeks.org/bubble-sort-algorithm/

Seguido de esta frase podemos determinar que los elementos adyacentes son los que intercambian posiciones, repetitivamente hasta que ya no haya cambios y así la lista queda ordenada.

Mejor caso: O(n) cuando la lista ya está ordenada, porque solo necesita una pasada para verificar que no hay swaps.

Peor caso: $O(n^2)$, ya que cada elemento puede necesitar compararse con todos los demás

Uso recomendado: Solo para fines educativos o cuando se trabaja con listas muy pequeñas y simples.

Ventajas: Fácil de entender e implementar.

Desventajas: Extremadamente ineficiente para listas grandes. Solo es útil si la lista está casi ordenada y se puede optimizar para terminar antes si no hay swaps en una pasada.

No recomendado para uso en producción o con grandes volúmenes de datos

Quick sort

"Selecciona un elemento como un pivote y particiona la matriz dada alrededor del pivote seleccionado colocando el pivote en su posición correcta en la matriz ordenada." (2025) https://www.geeksforgeeks.org/quick-sort-algorithm/

La elección del pivote puede variar, un elemento aleatorio o la mediana de la lista. La lista se reorganizará alrededor del pivote luego de iterarse recursivamente para volver a generar un pivote dentro del segmento que partió la lista principal, así se irá ordenando, dividiéndose en problemas más pequeños.

Mejor caso: O(n) cuando la lista ya está algo ordenada, y el pivote es bien elegido.

Peor caso: $O(n^2)$, si el pivote es mal elegido.

Uso recomendado: En listas medianas o extensas.

Ventajas: Cuando la lista es extensa y se necesita un ordenamiento rápido es muy buena elección, y elegir la mediana puede ser la opción más óptima. Es eficiente para ser usada con baja cantidad de memoria.

Desventajas: Según la elección del pivote, extremos, mediana o aleatorio, demora más, o bien termina cuando está todo ordenado, dejando ser un ordenamiento rápido. No funciona bien en listas pequeñas.

Selection sort

Su funcionamiento es muy simple, recorre la lista para encontrar el elemento más pequeño y lo intercambia por el primero, luego lo repite al proceso y lo intercambia por el segundo lugar, así repetitivamente, hasta terminar la lista. Está compuesto de dos bucles, uno para elegir el elemento de la lista y otro para la comparación de los elementos.

Mejor caso: $O(n^2)$ cuando la lista ya está ordenada.

Peor caso: O(nx), si la lista está ordenada a la inversa.

Uso recomendado: En listas medianas o extensas.

Ventajas: Es fácil de utilizarla e implementarla, utiliza poco espacio en memoria.

Desventajas: La complejidad temporal ya es más alta en el inicio que otras opciones como puede ser quick sort o merge sort.

Insertion sort

Esta forma de ordenamiento toma los elementos uno por uno desde la parte desordenada de la lista y los "inserta" en su posición correcta dentro de la porción de la lista que ya ha sido ordenada. Para poder hacer esto, comparará el elemento actual con los elementos ya ordenados, desplazándolos si es necesario para crear el espacio que le corresponde a la inserción.

Mejor caso: O(n) cuando la lista ya está ordenada.

Peor caso: $O(n^2)$, si la lista está ordenada a la inversa.

Uso recomendado: En listas pequeñas o medianas con alguna parte ya ordenada.

Ventajas: Es estable, ya que al existir dos elementos iguales el orden relativo se mantiene.

Desventajas: En listas grandes el rendimiento cae abruptamente porque requiere de muchos movimientos.

Merge sort

El principio fundamental es descomponer repetidamente una lista de elementos en sublistas más pequeñas hasta que cada sublista contenga un solo elemento. Luego el algoritmo comienza a "mezclarlas" o "fusionarlas" de nuevo de forma ordenada.

La mezcla se realiza comparando los elementos de dos sublistas ya ordenadas y combinándolas para formar una única sublista ordenada. Esta fusión se va repitiendo nivel por nivel, hasta que todas las sublistas se hayan combinado en una única lista final que será la ordenada.

Mejor caso: O(n) cualquier caso como esté la lista.

Peor caso: O(n) cualquier caso como esté la lista.

Uso recomendado: En listas extensas funciona muy bien al dividir los volúmenes.

Ventajas: Es estable, confiable y rendimiento constante, a su vez puede ir ejecutando “ramas” al mismo tiempo.

Desventajas: Necesita de una buena cantidad de memoria para almacenar los datos y realizar fusiones, no es eficiente en listas pequeñas.

Medición de algoritmos

Los algoritmos tienen una medición de la complejidad que es O(n) para determinar su eficiencia, cuanto tarda en resolver en función de la entrada. Se establece con n al tiempo que tarda en terminar una búsqueda de forma lineal, entonces ningún tipo de búsqueda tardará más que ella, el tiempo de esta búsqueda es directamente proporcional al tamaño de la lista.

Luego con la búsqueda binaria tendremos O(log n), lo cual es que tardará logarítmicamente el tamaño de la entrada, lo cual hace que si es del doble la entrada, tardará ligeramente más, podríamos decir, casi sin diferencia. Como sabemos por lo visto anteriormente para ejecutar una búsqueda binaria, la lista debe estar previamente ordenada.

Casos Prácticos

Análisis de algoritmos de búsqueda y de ordenamiento

Implementaremos un código donde crearemos una lista automática, de acuerdo a una cantidad establecida por el usuario, para luego hacer comparaciones de tiempo en los diferentes tipos de búsquedas, lineal y binaria, a su vez con diferentes tipos de ordenamientos, bubble sort, quick sort, selection sort e insertion sort.

Situación de la vida real donde podremos usar alguna de estas opciones

Ingresos en un estadio de fútbol

Planteamos un hipotético caso donde se necesitaría aplicar un algoritmo de búsqueda y de ordenamiento, en caso de usar la búsqueda binaria.

Este caso se trata del registro de las personas que fueron ingresando al estadio con su ID del carnet, estos son escaneados en la entrada registrando así el orden determinado día.

El club cuenta con 60.000 carnets emitidos, en el estadio ingresaron 55.541 personas.

De repente una persona preocupada llama al estadio preguntando si su hijo ese día ingresó en el estadio, ya que no puede comunicarse con él. El padre conoce el ID de su hijo, ya que posee una foto del carnet.

El encargado del club debe hacer la consulta de las personas ingresadas ese día y chequear si el hijo está dentro del estadio con el ID que le brindó el padre. Por ende el programa debe contar con algún algoritmo de búsqueda.

Metodología Utilizada

- Visual Studio Code
- Código Python
- GIT
- GitHub

Resultados Obtenidos

- Como resultado obtuvimos los tiempos de cada uno los tipos de búsqueda y ordenamientos, para poder determinar conclusiones

Conclusiones

La conclusión es, no hay un método que esté completamente bien o mal, cada uno de ellos se adapta con sus características a un determinado tipo de uso, es decir, en ciertos casos el desarrollador sabiendo el volumen de datos y la tarea que necesita optará por algunos de ellos, el que mejor se adapte a esa situación en particular, todos tienen su fuerte y sus desventajas, hay que saber cómo explotarlas y como determinar por cada uno de ellos.

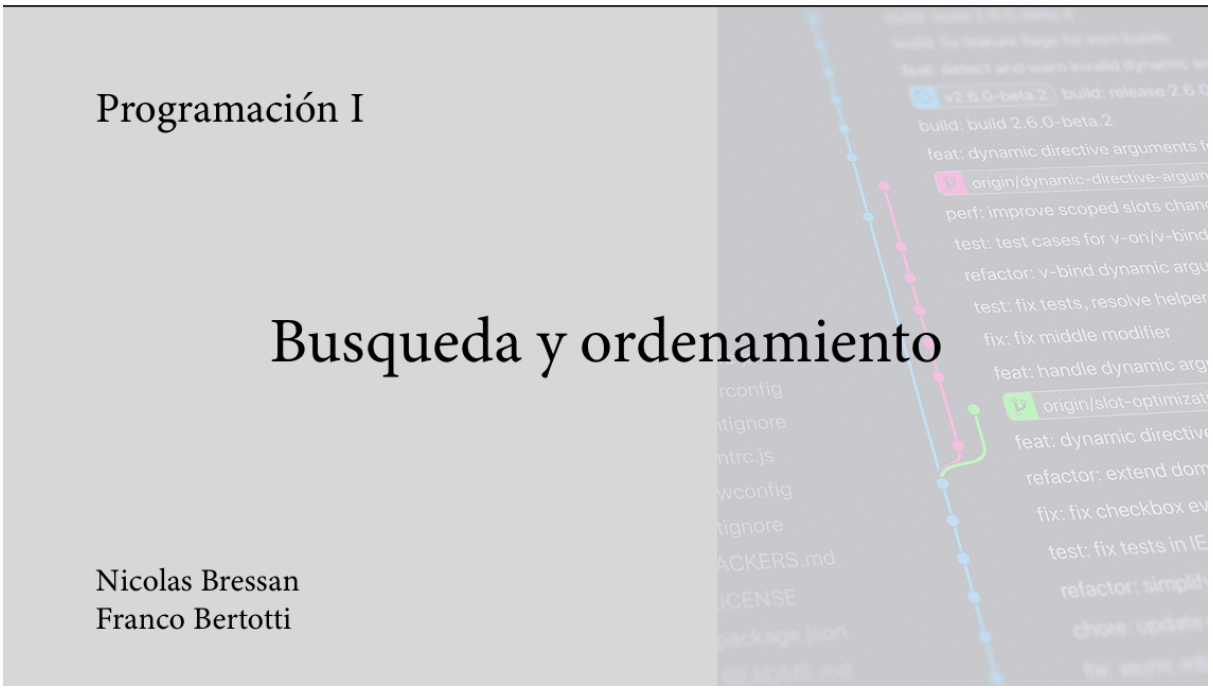
Bibliografía

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein. 3ra Edición the MIT press (2009) *Introduction to algorithms.*
- https://www.geeksforgeeks.org/searching-algorithms/
- https://www.geeksforgeeks.org/sorting-algorithms/
- https://www.programiz.com/dsa/binary-search
- https://www.programiz.com/dsa/bubble-sort
- https://www.youtube.com/watch?v=HmUpRHn31FU
- https://www.youtube.com/watch?v=9tZsDJ3JBUA
- https://www.geeksforgeeks.org/quick-sort-algorithm/

Anexos

Video explicativo: link
Repositorio: link

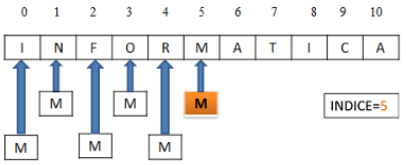
Imágenes:



UTN
UNIVERSIDAD TECNOLÓGICA NACIONAL

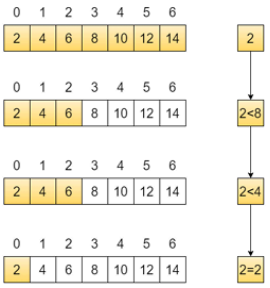
Busqueda lineal

-tiempo proporsional
-lenta
-listas pequeñas
-no ordenado



Busqueda binaria

-tiempo logarítmico
-rápida y efectiva
-listas pequeñas y grandes
-ordenado



Busqueda de inter-polación

-más eficiente que la binaria
-conjuntos grandes
-ordenado

Busqueda de hash

-tabla hash
-elemento con unica ubicación
-conjuntos grandes

Otros tipos de busqueda

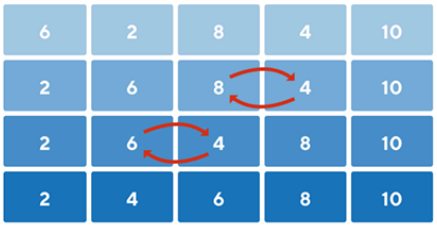
-palabras
-rutas
-documentos

TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

UTN
UNIVERSIDAD TECNOLÓGICA NACIONAL

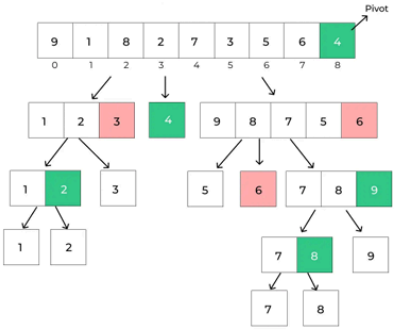
Ordenamiento Bubble sort

-comparación adyacente
-listas pequeñas



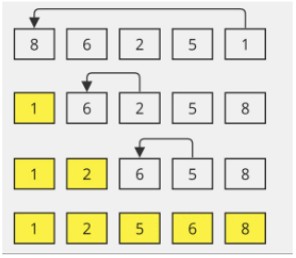
Quick sort

-pivote
-más eficiente que bubble



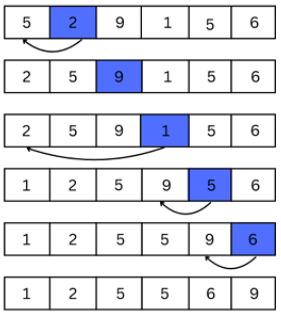
Selection sort

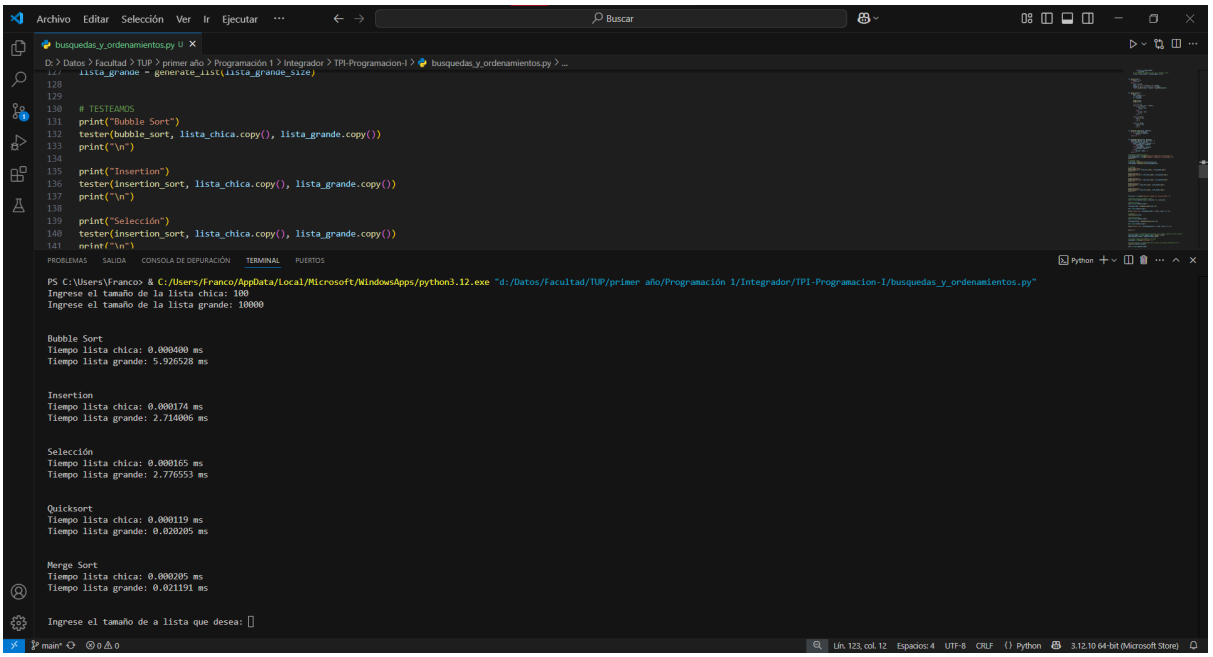
-elemento mas pequeño
-listas pequeñas

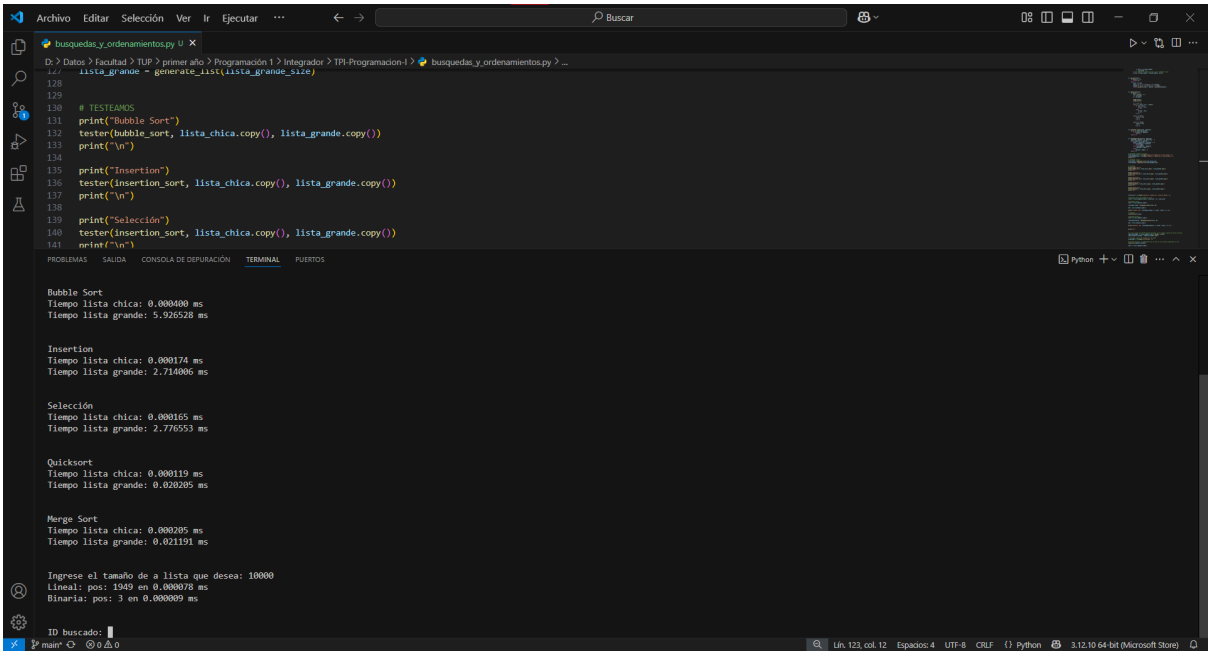


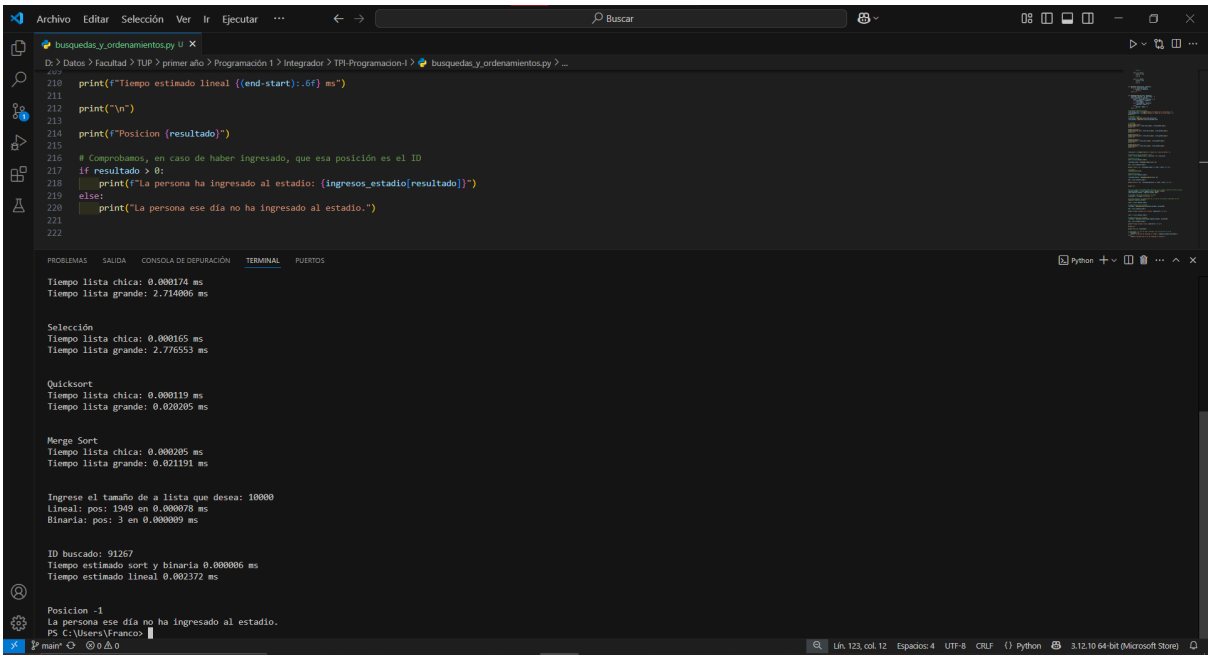
Insertion sort

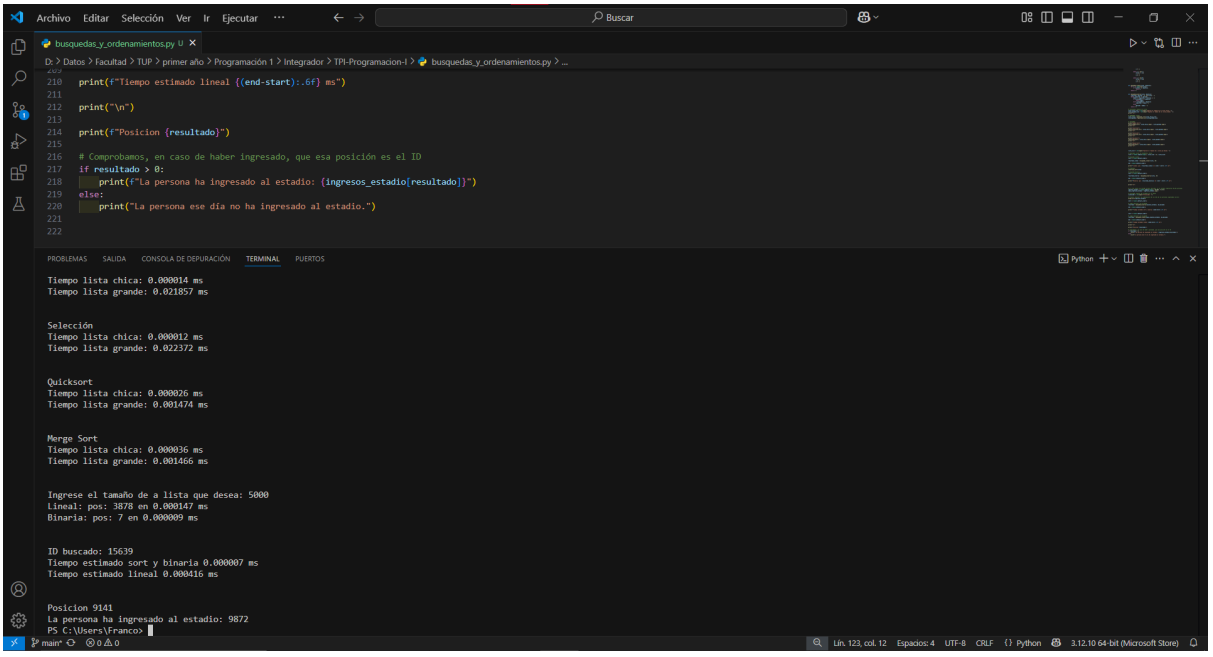
-elemento por elemento en su lugar
-listas pequeñas o algo ordenadas




Archivo Editar Selección Ver Ir Ejecutar
Buscar
busquedas_y_ordenamientos.py U
D: > Datos > Facultad > TUP > primer año > Programación 1 > Integrador > TPI-Programacion-I > busquedas_y_ordenamientos.py > ...
lista_grande = generate_list(lista_grande_size)
# TESTEAMOS
print("Bubble Sort")
tester(bubble_sort, lista_chica.copy(), lista_grande.copy())
print("\n")
print("Insertion")
tester(insertion_sort, lista_chica.copy(), lista_grande.copy())
print("\n")
print("Selección")
tester(insertion_sort, lista_chica.copy(), lista_grande.copy())
print("\n")
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
Python
PS C:\Users\Franco> & C:/Users/Franco/AppData/Local/Microsoft/WindowsApps/python3.12.exe "d:/Datos/Facultad/TUP/primer año/Programación 1/Integrador/TPI-Programacion-I/busquedas_y_ordenamientos.py"
Ingrese el tamaño de la lista chica: 100
Ingrese el tamaño de la lista grande: 10000
Bubble Sort
Tiempo lista chica: 0.000400 ms
Tiempo lista grande: 5.926528 ms
Insertion
Tiempo lista chica: 0.000174 ms
Tiempo lista grande: 2.714006 ms
Selección
Tiempo lista chica: 0.000165 ms
Tiempo lista grande: 2.776553 ms
Quicksort
Tiempo lista chica: 0.000119 ms
Tiempo lista grande: 0.020205 ms
Merge Sort
Tiempo lista chica: 0.000205 ms
Tiempo lista grande: 0.021191 ms
Ingrese el tamaño de a lista que desea:
main* 0 0
Lín. 123, col. 12 Espacios: 4 UTF-8 CRLF Python 3.12.10 64-bit (Microsoft Store)


Archivo Editar Selección Ver Ir Ejecutar
Buscar
busquedas_y_ordenamientos.py U
D: > Datos > Facultad > TUP > primer año > Programación 1 > Integrador > TPI-Programacion-I > busquedas_y_ordenamientos.py > ...
lista_grande = generate_list(lista_grande_size)
# TESTEAMOS
print("Bubble Sort")
tester(bubble_sort, lista_chica.copy(), lista_grande.copy())
print("\n")
print("Insertion")
tester(insertion_sort, lista_chica.copy(), lista_grande.copy())
print("\n")
print("Selección")
tester(insertion_sort, lista_chica.copy(), lista_grande.copy())
print("\n")
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
Python
Bubble Sort
Tiempo lista chica: 0.000400 ms
Tiempo lista grande: 5.926528 ms
Insertion
Tiempo lista chica: 0.000174 ms
Tiempo lista grande: 2.714006 ms
Selección
Tiempo lista chica: 0.000165 ms
Tiempo lista grande: 2.776553 ms
Quicksort
Tiempo lista chica: 0.000119 ms
Tiempo lista grande: 0.020205 ms
Merge Sort
Tiempo lista chica: 0.000205 ms
Tiempo lista grande: 0.021191 ms
Ingrese el tamaño de a lista que desea: 10000
Lineal: pos: 1949 en 0.000078 ms
Binaria: pos: 3 en 0.000009 ms
ID buscado:
main* 0 0
Lín. 123, col. 12 Espacios: 4 UTF-8 CRLF Python 3.12.10 64-bit (Microsoft Store)


Archivo Editar Selección Ver Ir Ejecutar
Buscar
busquedas_y_ordenamientos.py U
D: > Datos > Facultad > TUP > primer año > Programación 1 > Integrador > TPI-Programacion-I > busquedas_y_ordenamientos.py > ...
print(f"Tiempo estimado lineal {(end-start):.6f} ms")
print("\n")
print(f"Posicion {resultado}")
# Comprobamos, en caso de haber ingresado, que esa posición es el ID
if resultado > 0:
print(f"La persona ha ingresado al estadio: {ingresos_estadio[resultado]}")
else:
print("La persona ese día no ha ingresado al estadio.")
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
Python
Tiempo lista chica: 0.000174 ms
Tiempo lista grande: 2.714006 ms
Selección
Tiempo lista chica: 0.000165 ms
Tiempo lista grande: 2.776553 ms
Quicksort
Tiempo lista chica: 0.000119 ms
Tiempo lista grande: 0.020205 ms
Merge Sort
Tiempo lista chica: 0.000205 ms
Tiempo lista grande: 0.021191 ms
Ingrese el tamaño de a lista que desea: 10000
Lineal: pos: 1949 en 0.000078 ms
Binaria: pos: 3 en 0.000009 ms
ID buscado: 91267
Tiempo estimado sort y binaria 0.000006 ms
Tiempo estimado lineal 0.002372 ms
Posicion -1
La persona ese día no ha ingresado al estadio.
PS C:\Users\Franco>
main* 0 0
Lín. 123, col. 12 Espacios: 4 UTF-8 CRLF Python 3.12.10 64-bit (Microsoft Store)


Archivo Editar Selección Ver Ir Ejecutar
Buscar
busquedas_y_ordenamientos.py U
D: > Datos > Facultad > TUP > primer año > Programación 1 > Integrador > TPI-Programacion-I > busquedas_y_ordenamientos.py > ...
print(f"Tiempo estimado lineal {(end-start):.6f} ms")
print("\n")
print(f"Posicion {resultado}")
# Comprobamos, en caso de haber ingresado, que esa posición es el ID
if resultado > 0:
print(f"La persona ha ingresado al estadio: {ingresos_estadio[resultado]}")
else:
print("La persona ese día no ha ingresado al estadio.")
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
Python
Tiempo lista chica: 0.000014 ms
Tiempo lista grande: 0.021857 ms
Selección
Tiempo lista chica: 0.000012 ms
Tiempo lista grande: 0.022372 ms
Quicksort
Tiempo lista chica: 0.000026 ms
Tiempo lista grande: 0.001474 ms
Merge Sort
Tiempo lista chica: 0.000036 ms
Tiempo lista grande: 0.001466 ms
Ingrese el tamaño de a lista que desea: 5000
Lineal: pos: 3878 en 0.000147 ms
Binaria: pos: 7 en 0.000009 ms
ID buscado: 15639
Tiempo estimado sort y binaria 0.000007 ms
Tiempo estimado lineal 0.000416 ms
Posicion 9141
La persona ha ingresado al estadio: 9872
PS C:\Users\Franco>
main* 0 0
Lín. 123, col. 12 Espacios: 4 UTF-8 CRLF Python 3.12.10 64-bit (Microsoft Store)