

|  |
| --- |
| Programación I |
| Trabajo Integrador: Algoritmos de Búsqueda y Ordenamiento en Python |
| Universidad Tecnológica Nacional  Tecnicatura Universitaria en Programación a Distancia |
| Docente Titular Docente Tutor  Prof. Cinthia Rigoni Prof. Martín A. García |

Integrante

Bravo Molins Daiana Mariel

[daiana.bravo@tupad.utn.edu.ar](mailto:daiana.bravo@tupad.utn.edu.ar)

Fecha 17-06-2025

Índice

Introducción 3

Marco Teórico 4

Caso Practico 11

Metodología Utilizada 18

Resultados Obtenidos 19

Conclusiones 20

Bibliografía 22

Anexos 23

Introducción

La elección del tema, si bien no era de conocimiento previo, se vincula directamente con uno de los contenidos abordados en clase: las listas.  
En el presente trabajo se investigan los conceptos fundamentales de búsqueda y ordenamiento de datos, utilizando como fuente principal libros digitales disponibles en la web y la documentación oficial de Python.

La búsqueda y el ordenamiento son procesos fundamentales en programación, ya que permiten gestionar datos de forma eficiente, facilitando su acceso, análisis y utilización. En un contexto donde la información digital crece constantemente, disponer de mecanismos que agilicen la localización de datos específicos o que organicen grandes volúmenes de información resulta crucial.

Los algoritmos de búsqueda permiten encontrar elementos dentro de estructuras como listas, arreglos o bases de datos. Son vitales en aplicaciones cotidianas como buscadores web, filtros en plataformas, validación de entradas o recuperación de registros. Por otro lado, los algoritmos de ordenamiento reordenan los datos según un criterio determinado (alfabético, numérico, cronológico, etc.), mejorando la legibilidad, el procesamiento posterior y la eficiencia de otros algoritmos, como los de búsqueda o compresión.  
El objetivo principal es comprender estos algoritmos y su funcionamiento, para luego aplicarlos en un caso práctico desarrollado en el lenguaje de programación Python, integrando así los saberes teóricos y prácticos adquiridos.

Marco Teórico

¿Qué es un algoritmo?

Un algoritmo es un conjunto de instrucciones para completar una tarea, enfocado en soluciones eficientes. No se evalúa solamente cuánto tarda en ejecutarse, sino **cómo crece el número de operaciones necesarias** a medida que aumenta el tamaño del problema.

¿Programa y algoritmo es lo mismo?

No. Un programa es una serie de instrucciones ordenadas, codificadas en un lenguaje de programación que expresa uno o varios algoritmos y que puede ser ejecutado en una computadora.

Por lo general un programa contempla un conjunto de instrucciones encargadas de controlar el flujo de ejecución del programa –a través de una interfaz que puede ser de consola o gráfica– y hacer las llamadas a los distintos algoritmos que lo forman.

¿Por qué se ordenan y buscan datos?

Los datos se ordenan, básicamente por dos razones, primero es mucho más fácil para una persona poder leerlos y segundo facilita poder buscar un elemento dentro de dichos datos.

**Algoritmos de Búsqueda**

Los Algoritmos de Búsqueda tienen como objetivo encontrar un valor específico dentro de una estructura de datos, como una lista, arreglo, diccionario o base de datos.

“Una búsqueda es un algoritmo que toma una colección y un elemento de destino y determina si el objetivo está en la colección, a menudo devolviendo el índice del objetivo.“ (think Python, ,pág. 227)

Búsqueda lineal

La búsqueda lineal (o secuencial) consiste en recorrer uno por uno los elementos de una colección hasta encontrar el que cumple con un criterio dado.  
Es el algoritmo más simple, pero **ineficiente en listas grandes**. Su **complejidad es O(n)**.

Búsqueda binaria

La búsqueda binaria se basa en comparar el elemento del medio entre dos extremos. Primero se evalúa el elemento en el medio del arreglo, es decir entre el primer y último elemento, si este elemento es mayor al buscado entonces se busca entre el primero y el anterior al consultado, sino entre el siguiente y el ultimo y así sucesivamente hasta encontrar el elemento deseado. Se puede utilizar cuando un conjunto de datos se ordena y almacena en forma secuencial. Es muy eficiente y su complejidad es O(log n).

“(…) Podríamos simplemente codificar ambos algoritmos y probarlos en listas de distintos tamaños para ver cuánto tarda la búsqueda. Ambos algoritmos son bastante cortos, por lo que no sería difícil realizar algunos experimentos. Cuando probé los algoritmos en mi ordenador (un portátil algo anticuado), la búsqueda lineal fue más rápida para listas de longitud 10 o inferior, y no hubo mucha diferencia notable en el rango de longitud 10-1000. Después de eso, la búsqueda binaria fue la clara ganadora. Para una lista de un millón de elementos, la búsqueda lineal tardó un promedio de 2,5 segundos en encontrar un valor aleatorio, mientras que la búsqueda binaria solo tardó un promedio de 0,0003 segundos.” ( John Zelle, 2016,pag 369)

“Cada vez que hablo de un algoritmo, hablo de su tiempo de ejecución. Generalmente, conviene elegir el algoritmo más eficiente. Ya sea que esté tratando de optimizar el tiempo o el espacio. Volviendo a la búsqueda binaria. ¿Cuánto tiempo se ahorra usándola? Bueno, el primer método fue comprobar cada número uno por uno. Si se trata de una lista de 100 números, se necesitan hasta 100 intentos.

Si se trata de una lista de 4 mil millones de números, se requieren hasta 4 mil millones de intentos. Por lo tanto, el número máximo de intentos es igual al tamaño de la lista.

Esto se llama tiempo lineal.

La búsqueda binaria es diferente. Si la lista tiene 100 elementos, se necesitan como máximo 7 intentos. Si la lista tiene 4 mil millones de elementos, se necesitan como máximo 32 intentos. Potente, ¿verdad? La búsqueda binaria se ejecuta en tiempo logarítmico (o tiempo logarítmico, como lo llaman los nativos). Aquí tienes una tabla que resume nuestros hallazgos de hoy.

La notación Big O es una notación especial que indica qué tan rápido es un algoritmo.”

( Bhargava, 2016, pág. 29)

Existen otros tipos de búsqueda (aunque son más avanzados):

- **Por hashing:**

No se recorre ni divide, se usa una **clave (key)** para acceder directamente al valor. Es muy rápida.

#### **-por interpolación:**

#### Parecida a la binaria, pero en lugar de cortar a la mitad, **estima** dónde podría estar el valor. Solo es útil en listas **numéricas muy grandes y bien distribuidas**. Poco usada en Python, más común en bajo nivel (Ej: C).

#### - **Búsqueda exponencial:**

#### Hace saltos exponenciales (1, 2, 4, 8, 16...) hasta pasarse del valor buscado, y luego usa búsqueda binaria en ese intervalo. Es eficiente en grandes volúmenes de datos.

#### **-Búsqueda por salto (Jump Search):**

Salta bloques fijos de elementos en una lista ordenada. Menos común que la binaria, pero también más rápida que la lineal en algunos casos

#### **Búsqueda en profundidad y en anchura (DFS/BFS):**

#### Se usan en **estructuras no lineales**, como **árboles o grafos**. Muy comunes en problemas de caminos, juegos, laberintos, inteligencia artificial.

### **Algoritmos de Ordenamiento**

#### Ordenamientos para fines educativos (no nativos en Python)

**Bubble Sort** (burbuja): O(n²) - Ordenamiento burbuja

Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Repite el proceso varias veces hasta que toda la lista esté ordenada.

* Ejemplo (lista: [5, 3, 1]):  
  Compara 5 y 3 → los intercambia → [3, 5, 1]  
  Compara 5 y 1 → los intercambia → [3, 1, 5]  
  Luego repite el proceso.

Es muy fácil de implementar, pero muy lento en listas grandes.

**Selection Sort**: O(n²) - Ordenamiento por selección

Busca el **mínimo valor de la lista** y lo coloca al principio. Luego repite el proceso con el resto de la lista.

* **Ejemplo** (lista: [3, 1, 5]):  
  Elige 1 → lo pone primero → [1, 3, 5]  
  Luego sigue con el subarreglo restante.

**Es** fácil de entender y usar, pero **ineficiente** para listas largas. Siempre hace la misma cantidad de comparaciones, aunque esté ordenada.

**Insertion Sort**: O(n²) - Ordenamiento por inserción

Toma los elementos uno por uno e **inserta cada uno en el lugar correcto** respecto a los anteriores (como ordenar naipes en la mano).

* **Ejemplo** (lista: [3, 1, 5]):  
  3 está ordenado.  
  1 va antes → [1, 3, 5]  
  5 ya está en su lugar.

**Es** eficiente en **listas pequeñas o casi ordenadas,** pero no es óptimo para grandes volúmenes de datos.

Estos algoritmos son útiles para entender la lógica de ordenamiento, pero **no se usan en producción** por su baja eficiencia.

#### Ordenamientos eficientes (usados en producción)

* **Merge Sort**: O(n log n) - Ordenamiento por mezcla

Divide la lista en mitades recursivamente hasta que cada parte tenga un solo elemento, y luego **las mezcla ordenadamente**.

* **Ejemplo** (lista: [4, 2, 7, 1]):  
  Se divide → [4, 2] y [7, 1]  
  Se divide otra vez → [4], [2], [7], [1]  
  Se mezclan ordenados → [2, 4] y [1, 7] → [1, 2, 4, 7]

**Es** muy eficiente y **estable** (no cambia el orden relativo de elementos iguales).

**Quick Sort**: O(n log n) promedio, O(n²) en el peor caso - Ordenamiento rápido

Selecciona un **elemento pivote** y divide la lista en dos partes: los menores al pivote y los mayores. Aplica recursivamente la misma idea.

* **Ejemplo** (lista: [5, 3, 7, 2], pivote: 5):  
  Menores: [3, 2]  
  Mayores: [7]  
  Se ordena recursivamente y se combinan: [2, 3, 5, 7]

**Es** muy rápido en la práctica. En el peor caso (si el pivote es siempre el menor o mayor), su rendimiento **baja a O(n²)**.

**Heap Sort**: O(n log n) – (Ordenamiento por montículo)

Convierte la lista en un **montículo (heap)**, una estructura de árbol binario. Luego **extrae el valor máximo** y lo coloca al final, repitiendo el proceso.

* **Ejemplo:** crea un heap con [4, 10, 3] → el mayor es 10  
  Lo saca y ordena → [4, 3], repite.

No necesita espacio extra como Merge Sort, pero **no es estable** y más complejo de implementar.

Timsort

Python usa **Timsort** en sus funciones nativas sort() y sorted().  
Timsort es una mezcla optimizada de Merge Sort e Insertion Sort. Es **estable**, muy rápido en la práctica, y su complejidad en el peor caso es **O(n log n)**.

Los algoritmos nativos ya están implementados dentro del lenguaje y **optimizados en C** (en CPython). Son rápidos, eficientes y fáciles de usar.

#### Búsqueda:

* in
* .index()
* .find()
* bisect (módulo estándar para búsqueda binaria)

Ordenamiento:

* .sort() (modifica la lista)
* sorted() (devuelve una nueva lista ordenada)

Estos métodos son los que **se usan en la vida real** para trabajar con listas en Python.

### **Algoritmos programables (no nativos)**

Estos algoritmos **no están integrados como funciones automáticas en Python**, pero pueden implementarse fácilmente desde cero. Son útiles para **aprender conceptos de lógica, estructuras de datos y eficiencia**.

Ejemplos:

* **Búsqueda lineal y binaria manual**
* **Bubble Sort**
* **Quick Sort**
* **Merge Sort**

Aunque no sean exclusivos de Python (también existen en Java, C, etc.), son **algoritmos universales** y muy importantes en la formación en programación.

### **Complejidad Algorítmica: Notación Big O**

La notación **Big O** se usa para describir cómo crece el tiempo de ejecución de un algoritmo en función del tamaño de la entrada.

* **O(n)**: tiempo lineal (cada elemento se revisa una vez)
* **O(log n)**: tiempo logarítmico (muy rápido, como la búsqueda binaria)
* **O(n²)**: cuadrático, ineficiente en grandes listas

Caso práctico:

Utilizamos la función len() para obtener el total de elementos dentro de la lista.

print(f"La cantidad total de libros es de:",len(libros))

Se utilizó la función sorted() para obtener una nueva lista ordenada por título, sin modificar la original.  
En todos los casos, se empleó el módulo time para medir el tiempo de ejecución de cada función y evaluar su rendimiento en base al tiempo transcurrido.

inicio = time.time()

libros\_ordenados\_por\_titulo = sorted(libros, key= lambda x: x["título"].lower())

for libro in libros\_ordenados\_por\_titulo:

    print(libro["título"]) #Pido que se imprima la lista ordenada de titulos no el diciconario completo

fin = time.time()

print(f"Tiempo transcurrido: {fin - inicio:.6f} segundos para ordenar por titulo con ordenamiento nativo.")

Se implementó una función de **búsqueda lineal** que recorre uno por uno los elementos de la lista de libros hasta encontrar el que coincide con el título buscado.  
Para hacer la comparación, se convirtió el título ingresado y el de cada libro a minúsculas con .lower(), asegurando así que la búsqueda no sea sensible a mayúsculas o minúsculas.

def busqueda\_lineal\_libros(t):

    for libro in libros:

        if libro["título"].lower() == t.lower():

            return libro

inicio = time.time()  # Guarda el tiempo actual en segundos

print(busqueda\_lineal\_libros("1984"))

fin = time.time()  # Tiempo después de la búsqueda

print(f"Tiempo transcurrido: {fin - inicio:.6f} segundos para buscar por titulo con buqueda lineal.")

Se implementó una función llamada busqueda\_binaria\_libros para buscar un libro en una lista ordenada de diccionarios (libros\_ordenados\_por\_titulo).

Se recibe dos parámetros: x (la lista ordenada de libros) e y (el título del libro que se desea buscar).

Se definen dos índices: izq (inicio de la lista) y der (final de la lista)

Mientras izq sea menor o igual a der, calcula la posición media (medio) del intervalo actual.

Se compara el título del libro en la posición media con el título buscado (y), ambos convertidos a minúsculas para evitar problemas con mayúsculas.

Si coinciden, retorna ese libro. Si el título medio es menor que el buscado, descarta la mitad izquierda ajustando izq. Si es mayor, descarta la mitad derecha ajustando der. Si no encuentra el libro, retorna None.

inicio = time.time()

def busqueda\_binaria\_libros(x,y):

    izq = 0

    der = len(x) -1

    while izq <= der:

        medio = (izq + der)//2

        titulo\_actual = libros\_ordenados\_por\_titulo[medio]["título"].lower()

        if titulo\_actual == y.lower():

            return x[medio]

        elif titulo\_actual < y.lower():

            izq = medio + 1

        else:

            der = medio - 1

    return None

resultado2 = busqueda\_binaria\_libros(libros\_ordenados\_por\_titulo, "La estrella más brillante del cielo")

print(resultado2)

fin = time.time()

print(f"Tiempo transcurrido: {fin - inicio:.6f} segundos para buscar por titulo con busqueda binaria.")

Se implementó la función bubble\_sort\_libros para ordenar la lista de libros por título usando el algoritmo de **ordenamiento burbuja**.

Se recorre la lista varias veces (100 iteraciones en este caso, que es la cantidad de libros).

En cada pasada se comparan pares de elementos adyacentes, en este caso los títulos de los libros convertidos a minúsculas para evitar problemas con mayúsculas.

Si un título está "mayor" que el siguiente (alfabéticamente), intercambia ambos libros de posición. De este modo, en cada iteración, el elemento más "grande" (según orden alfabético) "burbujea" hacia el final de la lista. El proceso se repite hasta que toda la lista queda ordenada.

def bubble\_sort\_libros(lista):

    for i in range(100):

        for j in range(0, 100-i-1):

            if lista[j]["título"].lower() > lista[j+1]["título"].lower():

                lista[j], lista[j+1] = lista[j+1], lista[j]

inicio = time.time()

bubble\_sort\_libros(libros)

for libro in libros:

    print(libro["título"])

fin = time.time()

print(f"Tiempo transcurrido: {fin - inicio:.6f} segundos para ordenar por titulo con ordenamiento por burbuja.")

Se implementó la función insertion\_sort\_libros para ordenar la lista de libros por título usando el algoritmo de ordenamiento por inserción.

Comienza desde el segundo elemento (índice 1) y lo considera como la "clave" a insertar en la parte ya ordenada a su izquierda.

Compara la clave con los elementos anteriores, moviendo cada uno que sea mayor hacia la derecha para dejar espacio. Inserta la clave en la posición correcta para mantener el orden. Repite este proceso para cada elemento hasta que toda la lista esté ordenada. La comparación se realiza en minúsculas para evitar errores por mayúsculas y minúsculas.

def insertion\_sort\_libros(lista):

    for i in range(1, len(lista)):

        clave = lista[i]

        j = i - 1

        while j >= 0 and clave["título"].lower() < lista[j]["título"].lower():

            lista[j + 1] = lista[j]

            j -= 1

        lista[j + 1] = clave

inicio = time.time()

insertion\_sort\_libros(libros)

for libro in libros:

    print(libro["título"])

fin = time.time()

print(f"Tiempo transcurrido: {fin - inicio:.6f} segundos para ordenar por titulo con ordenamiento por insercion.")

Se implementa un juego interactivo para adivinar el título de un libro a partir de pistas.

Se selecciona aleatoriamente un libro de la lista usando random.choice.

Se muestran varias pistas sobre el libro (autor, año de publicación, género literario, si tiene adaptación cinematográfica, nacionalidad del autor).

El usuario debe ingresar su respuesta con el título del libro. Se compara la respuesta del usuario (en minúsculas y sin espacios extra) con el título real (también en minúsculas).

Si coincide, muestra un mensaje de felicitación. Si no coincide, muestra cuál era el título correcto.

def adivina\_el\_libro(libros):

    libro = random.choice(libros)

    print("ADIVINÁ EL LIBRO CON ESTAS PISTAS")

    print(f"Autor: {libro["autor"]}")

    print(f"Año de publicación: {libro["año\_publicación"]}")

    print(f"Género: {libro["género\_literario"]}")

    print(f"¿Tiene adaptación cinematográfica?: {libro["adaptación\_cinematográfica"]}")

    print(f"Nacionalidad del autor: {libro["nacionalidad"]}")

    respuesta = input(f"¿Cual es el titulo del libro? ").strip().lower()

    if respuesta == libro["título"].lower():

        print("¡Correcto!! Usted se ha ganado Las obras completas de Nietzsche!")

    else:

        print(f"No, era: {libro['título']}")

adivina\_el\_libro(libros)

Segunda actividad lúdica: la función implementa un juego interactivo tipo **verdadero o falso** con afirmaciones relacionadas con libros y autores.

Primero, la función define una lista de afirmaciones, cada una asociada con la respuesta correcta:  
"v" para verdadero o "f" para falso.

Luego, selecciona aleatoriamente una afirmación usando random.choice. Muestra la afirmación y le pide al usuario que responda si es verdadera (v) o falsa (f). Si el usuario ingresa algo distinto a "v" o "f", le pide que ingrese una respuesta válida. Finalmente, compara la respuesta del usuario con la correcta y muestra si acertó o no.

def verdadero\_o\_falso(u):

    afirmaciones = [

        ("Un mundo feliz de Aldous Huxley fue publicado en 1932 y es una novela distópica.", "v"),

        ("Pedro Páramo es una saga de novelas mexicanas escritas por Juan Rulfo.", "f"),

        ("Demian de Hermann Hesse fue escrita originalmente en alemán y aborda temas de identidad y juventud.", "v"),

        ("Las intermitencias de la muerte de José Saramago fue publicada después de que el autor recibiera el Premio Nobel de Literatura.", "v"),

        ("El castillo de Franz Kafka es una obra póstuma y fue adaptada al cine.", "v"),

        ("La mecanografía de Pablo De Santis tiene una adaptación cinematográfica.", "f"),

        ("El Aleph es una novela corta escrita por Jorge Luis Borges.", "f"),

        ("El túnel de Ernesto Sabato es una novela corta psicológica que fue adaptada al cine.", "v"),

        ("La náusea de Jean-Paul Sartre fue publicada en francés y el autor aceptó el Premio Nobel de Literatura.", "f"),

        ("Hijos de los hombres de P.D. James es una novela distópica británica que fue adaptada al cine.", "v"),

        ("Travesuras de la niña mala de Mario Vargas Llosa ganó el Premio Nobel de Literatura.", "v"),

        ("Trafalgar de Benito Pérez Galdós es una novela histórica y forma parte de una saga.", "v"),

        ("Una habitación propia es un ensayo feminista escrito por Virginia Woolf en inglés.", "v"),

        ("El libro de los abrazos de Eduardo Galeano es una miscelánea que combina crónica, poesía y ensayo.", "v"),

        ("Las ciudades invisibles de Italo Calvino es una novela corta que fue adaptada al cine.", "f"),

        ("Todo se desmorona de Chinua Achebe es una novela histórica nigeriana publicada originalmente en inglés.", "v"),

        ("La carretera de Cormac McCarthy ganó el Premio Pulitzer y es una novela distópica estadounidense.", "v"),

        ("Orlando de Virginia Woolf fue publicada en 1928 y es una obra de ficción biográfica con temas feministas.", "v"),

        ("La piel del tambor de Arturo Pérez-Reverte no tiene adaptación cinematográfica.", "f"),

        ("La princesa prometida de William Goldman es una novela de fantasía y romance estadounidense adaptada al cine.", "v"),

        ("Ficciones de Jorge Luis Borges es una novela corta.", "f"),

        ("La historia interminable de Michael Ende es una novela fantástica alemana que fue adaptada al cine.", "v"),

        ("Los hombres me explican cosas es un ensayo feminista escrito por Rebecca Solnit.", "v"),

        ("Bajo la misma estrella de John Green es una novela juvenil romántica adaptada al cine.", "v"),

        ("Los renglones torcidos de Dios de Torcuato Luca de Tena aborda temas de locura y verdad y fue adaptada al cine.", "v")

        ]

    afirmacion, respuesta\_correcta = random.choice(afirmaciones)

    print("VERDADERO O FALSO")

    print("Afirmación:", afirmacion)

    respuesta = input("¿Verdadero (v) o Falso (f)? ").strip().lower()

    while respuesta != "v" and respuesta != "f":

        print("Debe responder con 'v' o 'f'.")

        respuesta = input("¿Verdadero (v) o Falso (f)?: ").strip().lower()

    if respuesta == respuesta\_correcta:

        print("¡Correcto!")

    else:

        print("Incorrecto.")

verdadero\_o\_falso(libros)

Se implemento una tercer actividad lúdica donde el usuario debe ingresar el titulo de un libro y el programa revisa si el mismo se encuentra en la lista. El programa recorre uno por uno los elementos de la lista y compara el título ingresado con los títulos existentes. Si encuentra una coincidencia, retorna el diccionario correspondiente al libro. En caso contrario, devuelve “None”.

# ======================= Busqueda lineal con adivinanza ========================

def busqueda\_lineal\_adivinanza(l, titulo\_buscado):

    for libro in l:

        if libro["título"].lower() == titulo\_buscado.lower():

            return libro

    return None

print("¡Adivina si esta el libro! Escribe el título completo:")

titulo\_usuario = input("Título: ").strip()

resultado = busqueda\_lineal\_adivinanza(libros, titulo\_usuario)

if resultado:

    print(f"¡Lo encontré! {resultado['título']} es de {resultado['autor']}")

else:

    print("No está en la lista. Intenta con otro título.")

busqueda\_lineal\_adivinanza(libros, titulo\_usuario)

Metodología utilizada:

Para la búsqueda de material de estudio para implementar en el código se buscó información en diferentes libros dedicados exclusivamente al lenguaje Python, disponibles en la web. Así como también paginas oficiales que ofrecieran explicaciones acompañadas de implementación de código.

Para poner en práctica los diferentes tipos de búsqueda y ordenamiento se utilizo el programa Python versión 3.13.3 trabajando en el editor de código Visual Studio Code.

En el desarrollo del código se empleó una **lista con múltiples diccionarios** (donde cada diccionario representa un libro con sus respectivos campos). A partir de esta misma estructura, se aplicaron diferentes métodos de búsqueda y ordenamiento, a fin de ponerlas en práctica por primera vez y analizar sus resultados.

Además, se implementó un juego interactivo como parte del trabajo, con el fin de incorporar un componente lúdico que favorezca el aprendizaje.

Resultados:

Durante la implementación del proyecto, se utilizó una estructura de datos compuesta por una lista con 100 diccionarios. Cada diccionario representaba un libro con distintos campos: título, autor, año de publicación, nacionalidad del autor, género literario y si tiene adaptación cinematográfica, entre otros campos.

Se aplicaron distintos algoritmos de búsqueda y ordenamiento con el objetivo de evaluar su funcionamiento, eficiencia y utilidad práctica. A continuación, se describen los principales resultados obtenidos:

Conteo de elementos:  
Se utilizó la función nativa len() para confirmar la cantidad de libros en la lista. El resultado fue correcto: 100 libros cargados.

Ordenamiento con sorted() (nativo):  
Se usó la función sorted() con una expresión lambda para ordenar alfabéticamente los títulos de los libros. El proceso fue rápido y efectivo, y permitió aplicar posteriormente la búsqueda binaria.

Búsqueda Lineal (manual):  
Se recorrió la lista buscando un título específico ("1984"). El algoritmo devolvió correctamente el diccionario completo del libro.

Búsqueda Binaria (manual):  
Sobre la lista previamente ordenada, se implementó una búsqueda binaria por título. El resultado fue exitoso: se encontró el libro deseado (“La estrella más brillante del cielo”) de forma eficiente.

Ordenamiento por Burbuja:  
Se implementó este algoritmo clásico para ordenar los libros por título. Aunque funcional, fue considerablemente más lento comparado con el método sorted().

Ordenamiento por Inserción:  
Otro algoritmo educativo, que resultó menos eficiente que el de burbuja en esta pequeña lista.

Actividad Lúdica: "Adivina el libro":  
Se programó una función interactiva que selecciona un libro al azar y muestra pistas al usuario (autor, año, género, etc.) para que intente adivinar el título.  
Esto permitió consolidar los datos de la lista y hacer el aprendizaje más dinámico y entretenido.

Actividad "Verdadero o Falso":  
Se incorporó un juego educativo con afirmaciones sobre libros, autores y géneros. El usuario debía responder si la frase era verdadera o falsa. Esto amplió el enfoque del trabajo y conectó los datos con conocimientos generales de literatura.

Actividad “Busqueda\_lineal\_adivinanza”:

Esta actividad tiene como objetivo aplicar el algoritmo de búsqueda lineal de manera sencilla y entretenida. El usuario debe adivinar si un libro se encuentra en la lista escribiendo su título completo. Esta actividad sirvió para reforzar el uso práctico de algoritmos simples como la búsqueda secuencial, al mismo tiempo que fomenta el interés por la literatura.

Conclusión:

Es importante comprender que **no existe una única estructura o algoritmo que sea universalmente el mejor**. Cada uno tiene sus **ventajas y desventajas** según el contexto. Por eso, resulta fundamental que el programador conozca cómo funciona cada algoritmo y sepa **cuándo conviene utilizar uno u otro**, incluso combinándolos si es necesario dentro de un mismo programa.

Por ejemplo, el **algoritmo de búsqueda binaria** resulta mucho más eficiente que la búsqueda lineal, ya que **reduce la zona de búsqueda a la mitad en cada paso**, acelerando notablemente el proceso. Esto se evidencia en los tiempos medidos en este trabajo:

* Búsqueda lineal: 0.000090 segundos
* Búsqueda binaria: **0.000079 segundos**

Aunque la diferencia en este caso parece mínima, en estructuras de datos mucho más grandes la mejora se vuelve considerable.

De igual forma, los algoritmos de ordenamiento muestran diferencias claras:

* sorted() (función nativa de Python optimizada con Timsort): **0.003552 s**
* Ordenamiento por burbuja: **0.005941 s**
* Ordenamiento por inserción: **0.015984 s**

Esto demuestra por qué los lenguajes de programación modernos, como Python, ya incluyen **algoritmos eficientes en sus bibliotecas estándar**, los cuales están optimizados y escritos en bajo nivel (como C). De hecho, muchos ordenadores y sistemas modernos utilizan combinaciones de algoritmos según la cantidad de datos o la estructura a ordenar. Timsort, por ejemplo, es una mezcla de Merge Sort e Insertion Sort que aprovecha lo mejor de ambos.

En definitiva, dominar estos algoritmos permite tomar mejores decisiones al programar, optimizando los tiempos de ejecución y aprovechando al máximo los recursos del sistema.

Bibliografía consultada:

- Bhargava, A. (2016). Grokking algorithms: An illustrated guide for programmers and other curious people. Manning Publications.

- Miller, B., & Ranum, D. (s.f.). Solución de problemas con algoritmos y estructuras de datos usando Python (M. Orozco-Alzate, Trad.). Luther College. Universidad Nacional de Colombia - Sede Manizales. https://runestone.academy/ns/books/published/pythonds/index.html

- Bel, W. (Trad.). (2014). Algoritmos y estructuras de datos en Python (B. N. Miller & D. L. Ranum, Autores). Pearson Educación.

- Downey, A. B. (2015). Think Python: How to think like a computer scientist (2.ª ed., versión 2.4.0). Green Tea Press. https://www.thinkpython.com

-Zelle, J. (2016). Python programming: An introduction to computer science (2nd ed.). Franklin, Beedle & Associates.

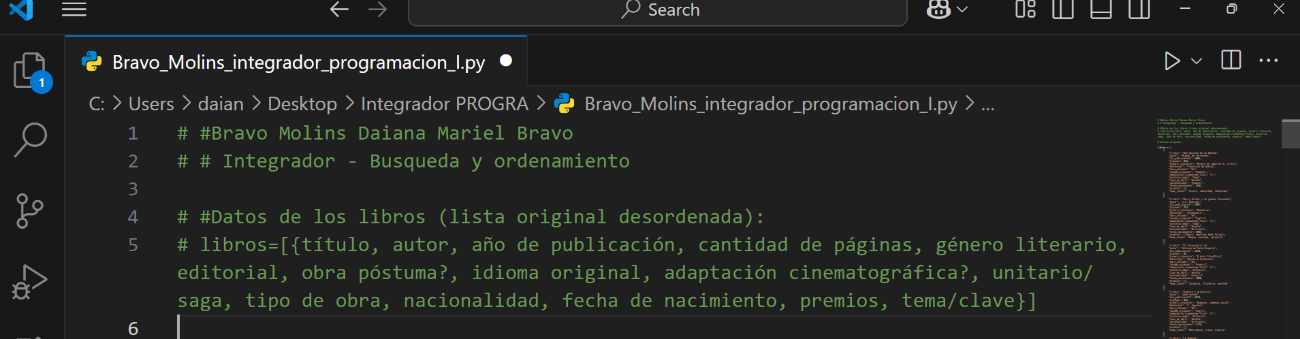
- https://docs.python.org/es/3.13/howto/sorting.html

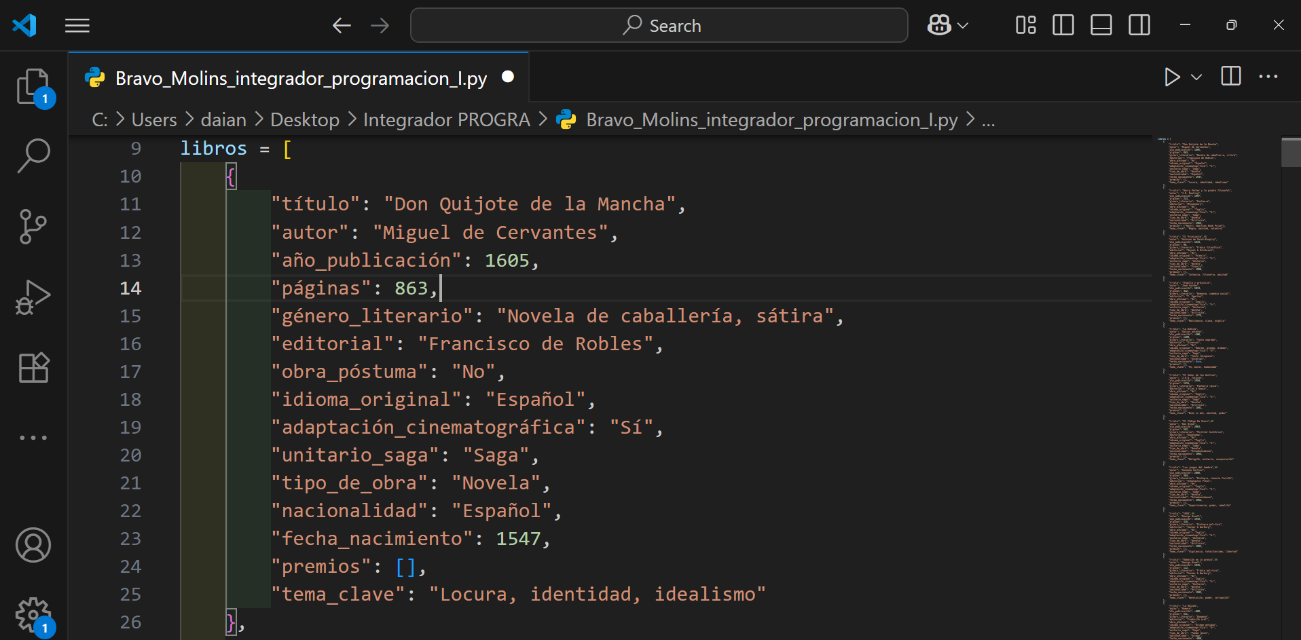
Anexos

Link a YouTube:

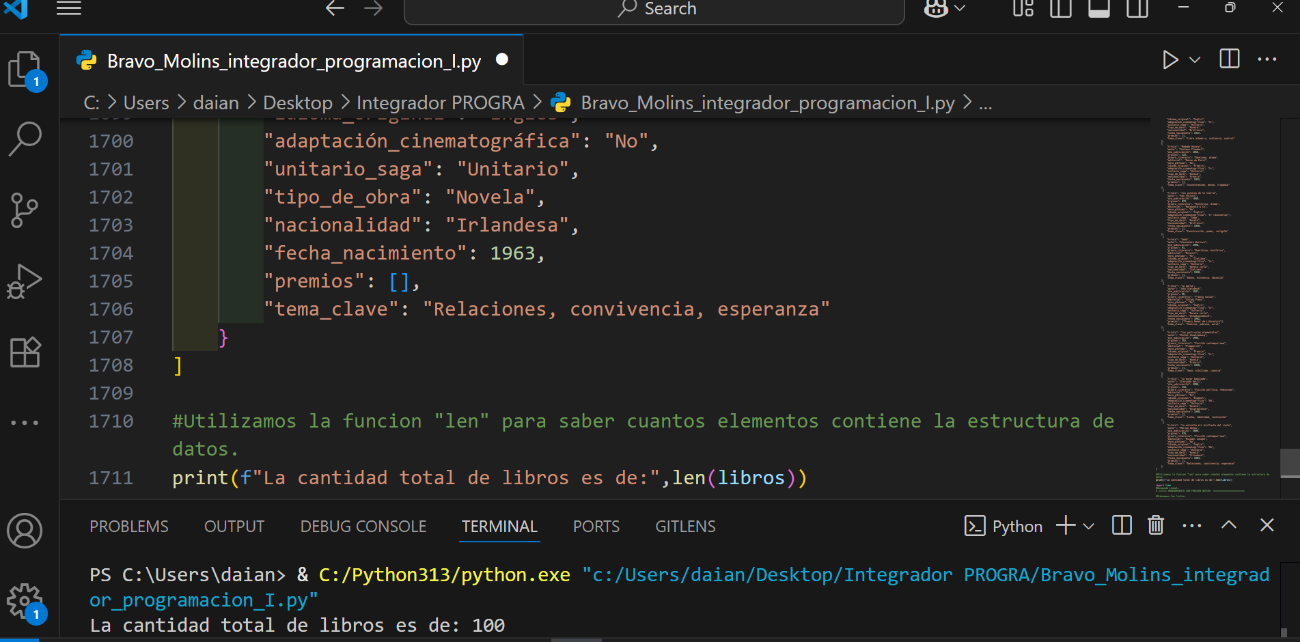
Capturas de pantalla con el código funcionando:

Datos utilizados en la lista:

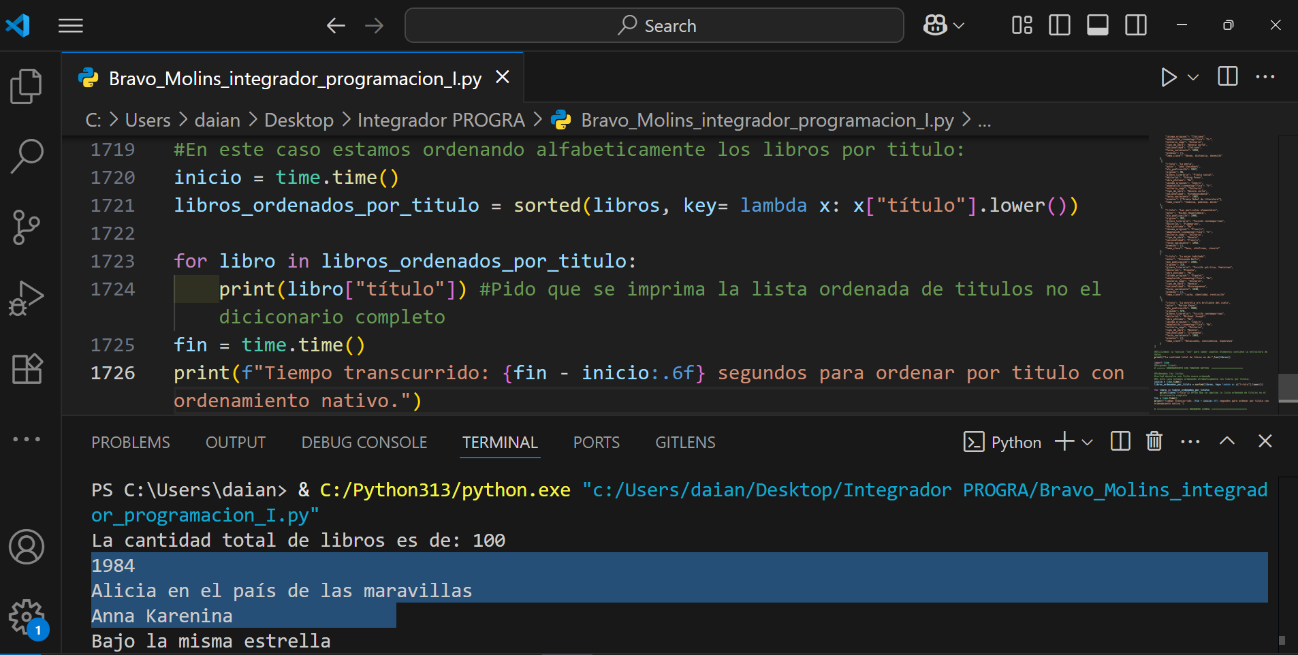




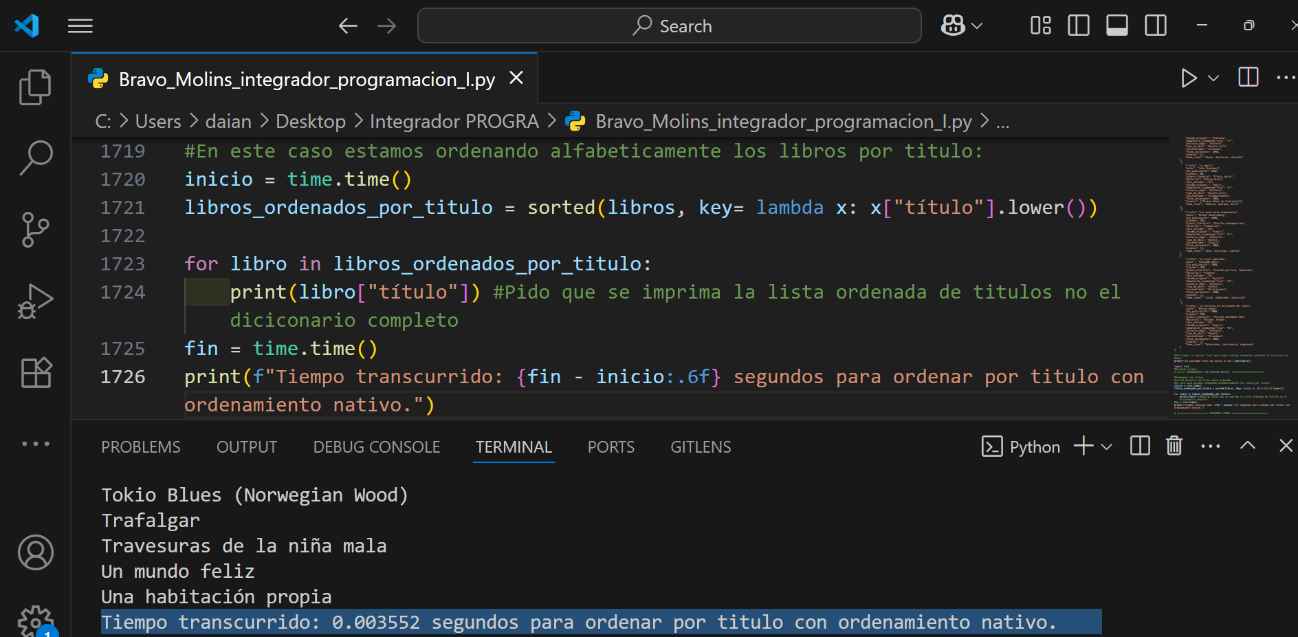
Función len():



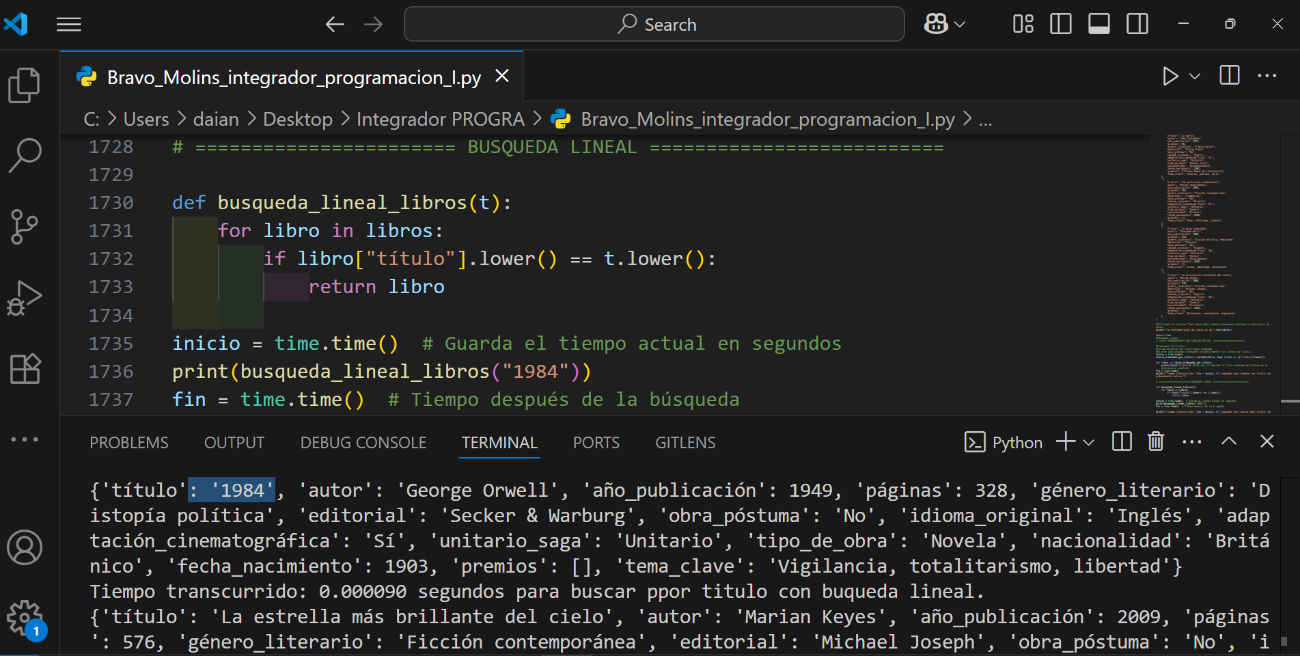
Función sorted():



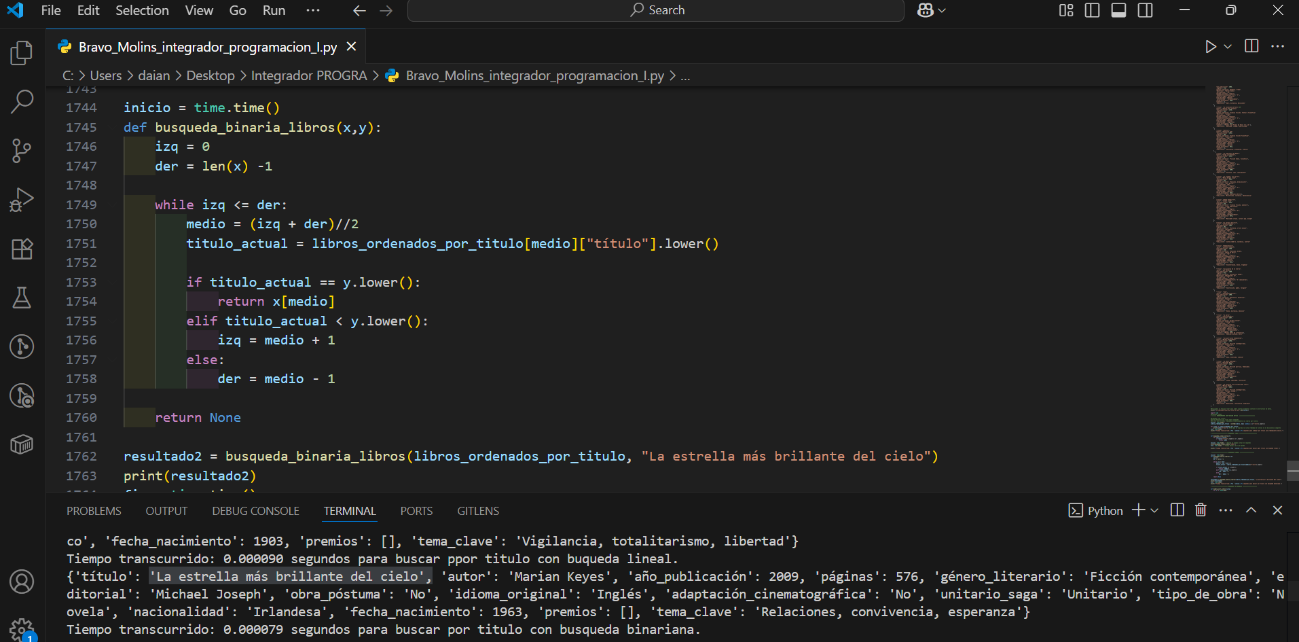
Medición del tiempo con el módulo time() usando sorted():



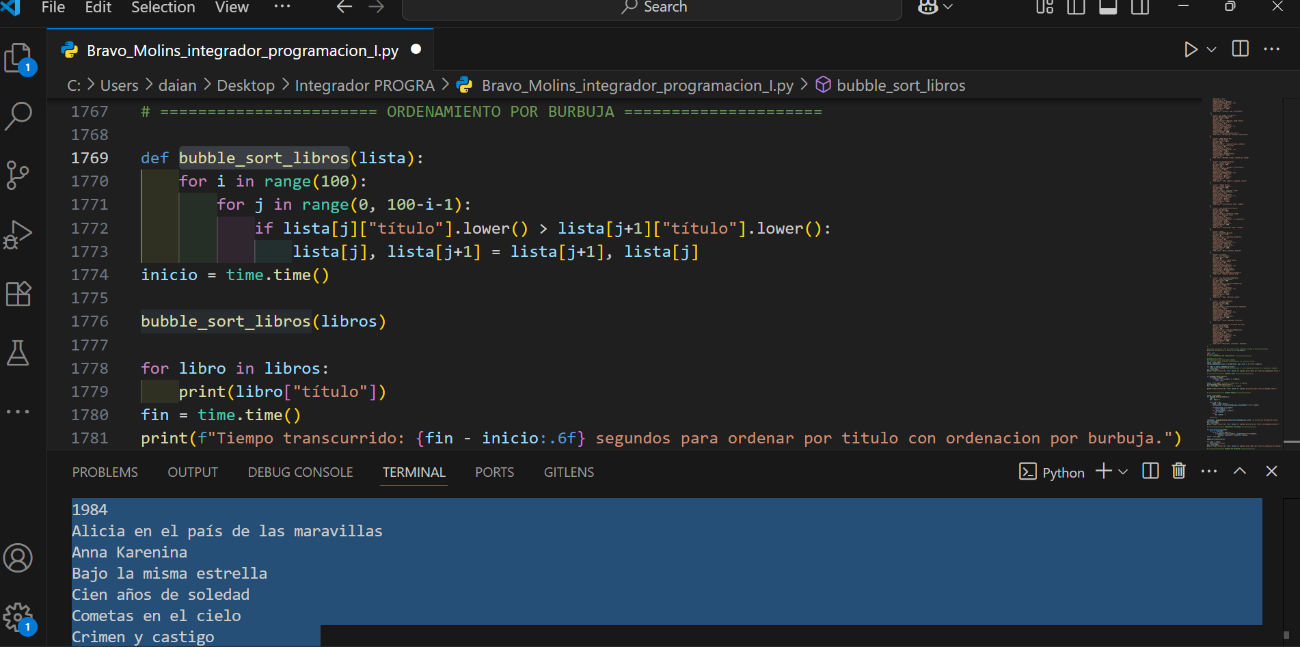
Búsqueda lineal(1984) más tiempo transcurrido:

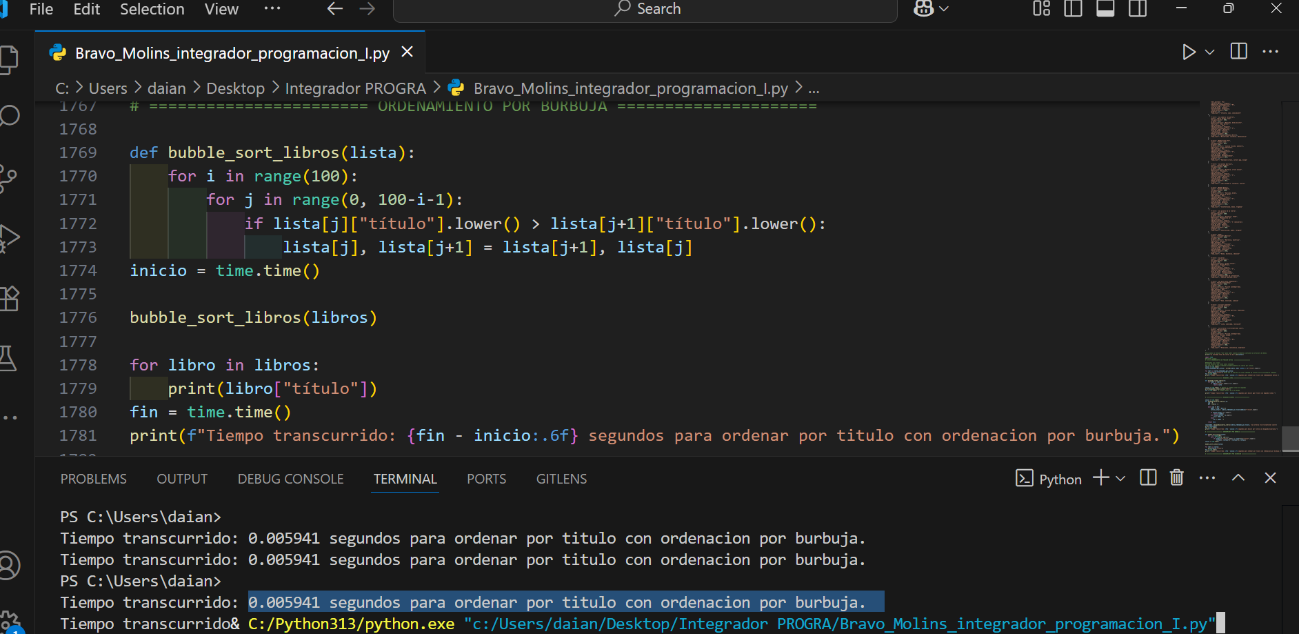


Búsqueda binaria(“La estrella más brillante del cielo”) más tiempo transcurrido:

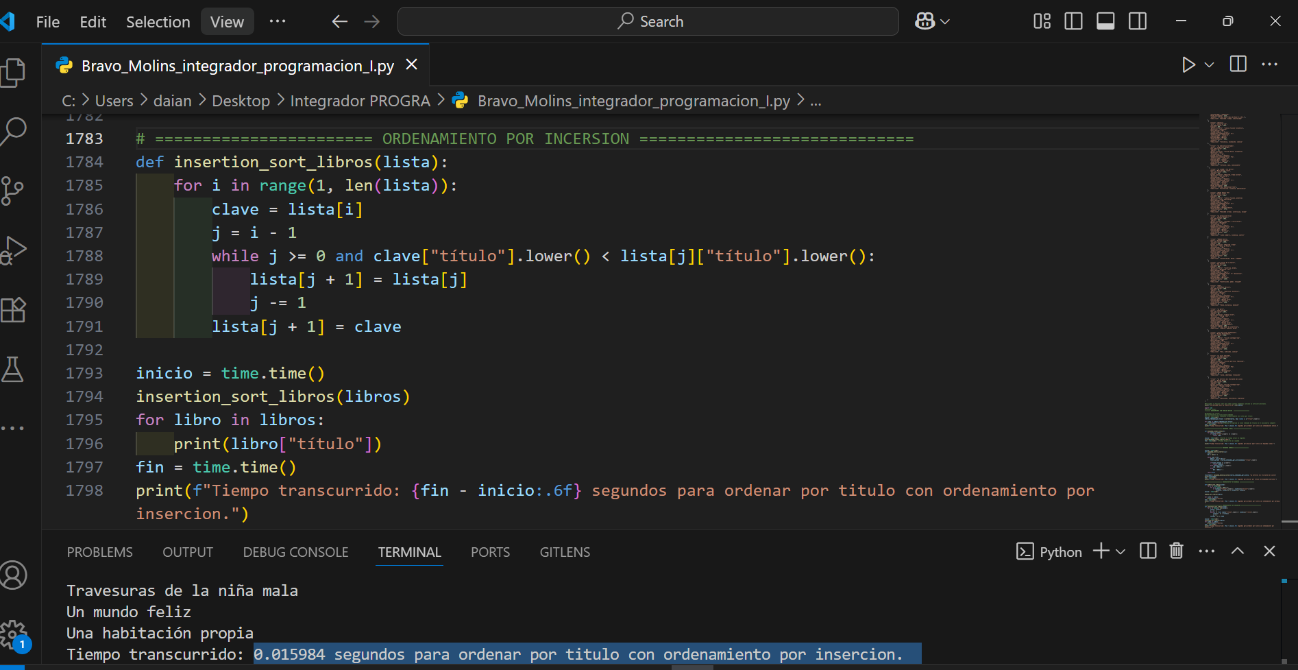


Ordenamiento por burbuja:

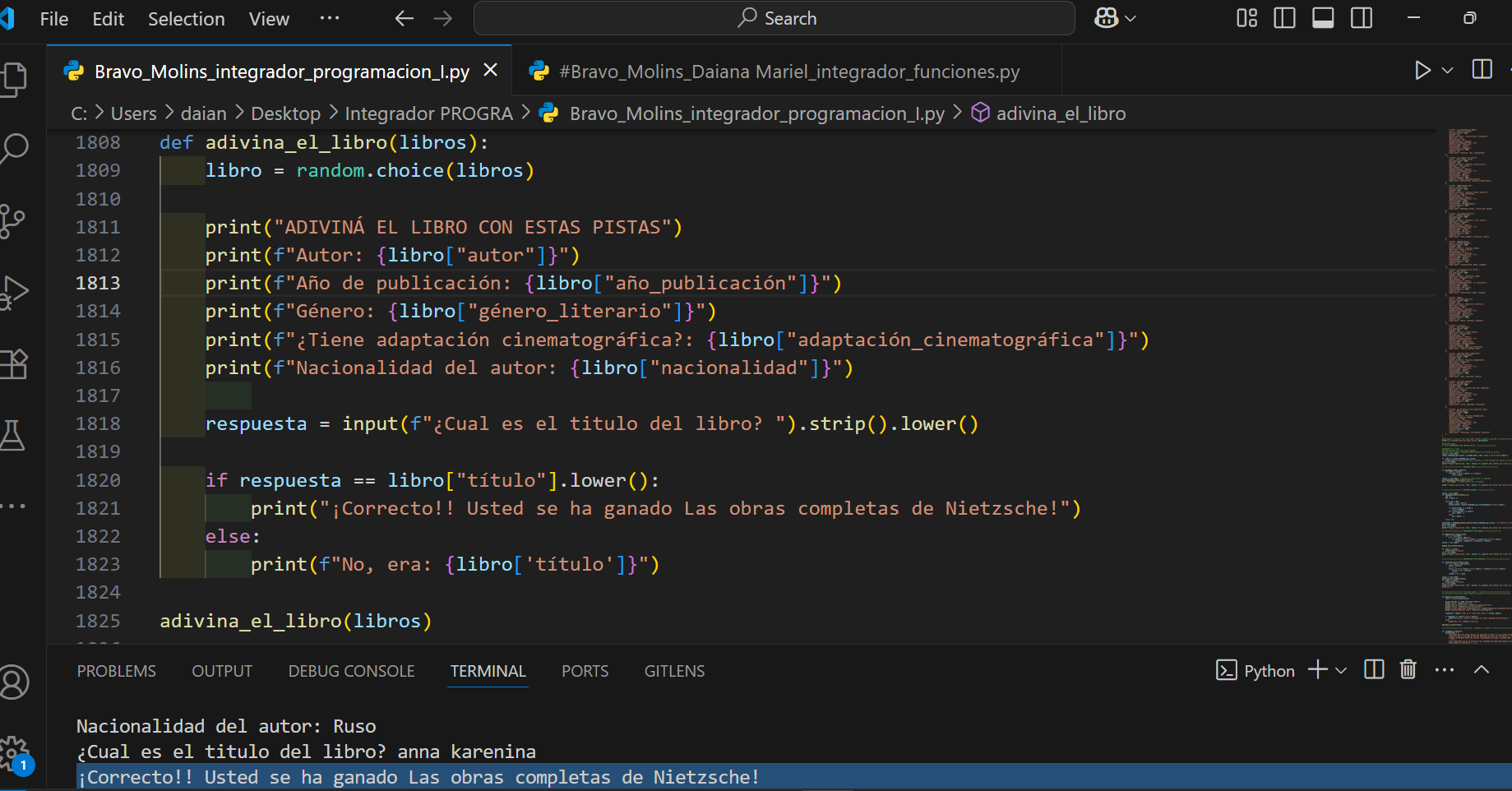




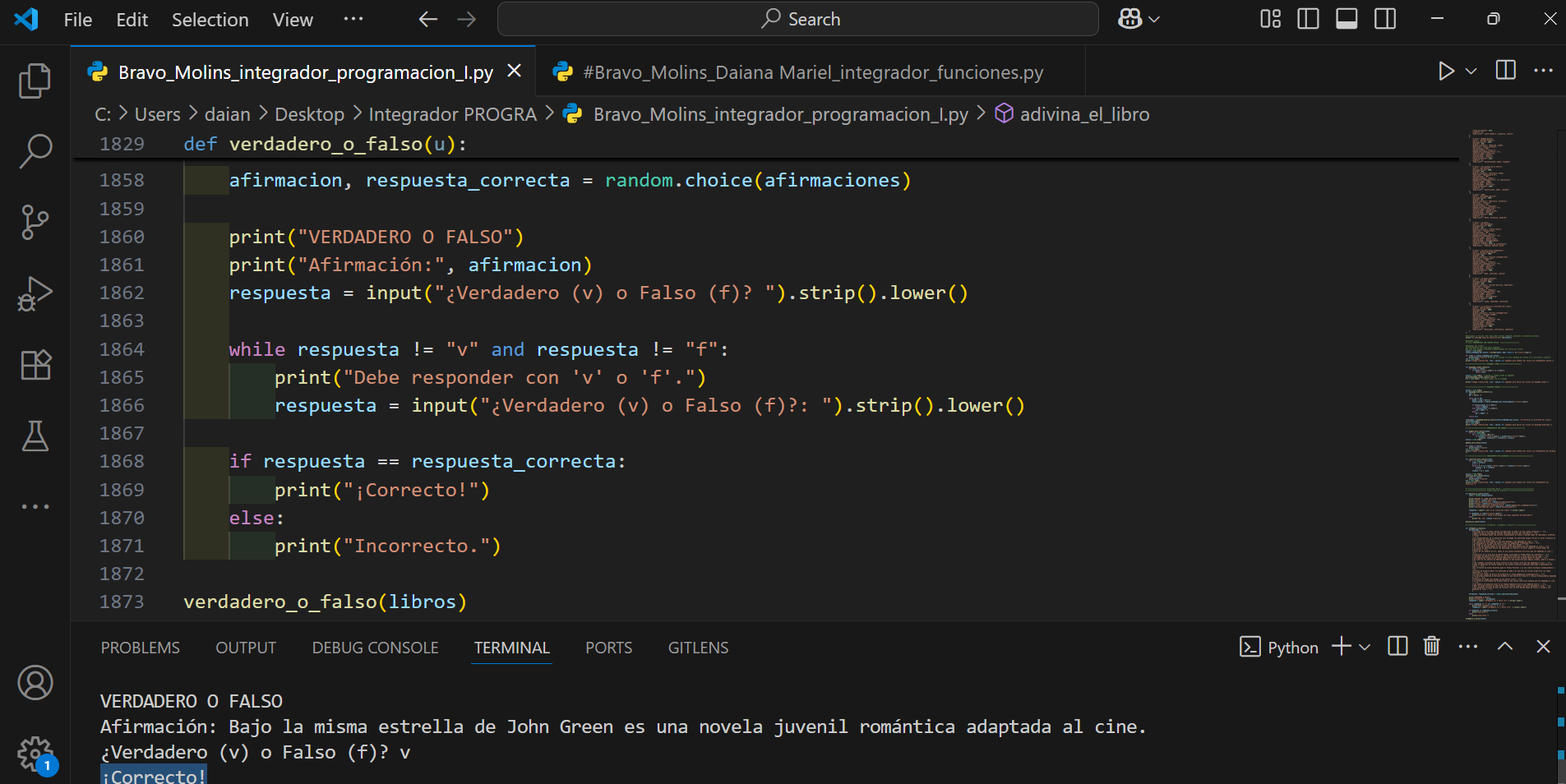
Ordenamiento por inserción más tiempo transcurrido:



Primer actividad lúdica (adivina el libro):



Segunda actividad lúdica (verdadero o falso):



Tercer actividad lúdica Búsqueda lineal con adivinanza:

