Algoritmo de búsqueda y ordenamiento – Quicksort

Integrantes:

- Basualdo Arcati Pablo
- Kaddarian Dante

Materia:

- Programación I

Profesor:

- Cinthia Rigoni

Fecha de entrega:

- Lunes 09/06/2025

Índice

Introducción 3

Marco teórico 4

Caso Práctico 6

Metodología Utilizada 7

Resultados obtenidos 8

Conclusiones 9

Bibliografía 10

Anexo 11

Introducción

Desde los primeros días del curso, la palabra “algoritmo” aparece una y otra vez. Algunos compañeros la entendieron enseguida. Otros, tardaron más. No porque sea un concepto difícil, sino porque su definición parece más técnica de lo que realmente es. Es algo que está en todo. A veces sin que lo notemos.

Un algoritmo es, básicamente, una serie de pasos. Pasos que permiten llegar de un punto a otro, resolver algo. En el libro de Introducción a los Algoritmos de Thomas Cormen[1], se habla de transformar datos de entrada en datos de salida, de forma clara. No siempre es tan claro cuando se está aprendiendo, pero la idea va quedando.

Lo que pasa es que, al principio, uno cree que con que funcione, ya está. Pero no. Más adelante se ve que importa también cómo lo hace. Cuánto tarda, cuántas veces repite algo. Ahí es donde se empieza a hablar de comparar distintos algoritmos, incluso si hacen lo mismo.

Uno de los ejemplos más vistos es cuando hay que buscar un dato. O cuando hay que ordenar una lista. Se trabajan por separado, pero están muy relacionados. Porque para buscar bien, muchas veces primero hay que ordenar. Y eso cambia todo. O por lo menos cambia cómo se piensa el problema.

Aunque un algoritmo esté bien planteado, su eficiencia puede variar mucho según el tipo de datos o la cantidad de elementos. Por eso, los casos de prueba y la observación del comportamiento real se vuelven tan importantes como la teoría. A veces es cuestión de milisegundos, otras veces de horas. Y cuando se trabaja con millones de datos, eso se nota. Y mucho.

En nuestro caso, además de entender qué es un algoritmo, tenemos que empezar a pensar en cómo se construyen en la práctica. Python, como lenguaje, nos da herramientas claras para eso. Con él vamos a poder implementar algunos de estos algoritmos y ver en código lo que ahora estamos viendo en teoría.

Esto es solo una primera mirada. En los próximos apartados vamos a enfocarnos especialmente en dos grupos clave: los algoritmos de búsqueda y los de ordenamiento. Ahí empieza otro tipo de análisis.

Marco teórico

Cuando uno empieza a programar, enseguida aparecen dos cosas: buscar datos y ordenarlos. A veces no se nota, pero están en casi todo. Desde encontrar un nombre en una lista, hasta mostrar resultados ordenados de menor a mayor. La verdad que entender estos procesos ayuda un montón a pensar cómo se trabaja con datos en general.

Un algoritmo es un conjunto de pasos que sirven para resolver un problema. En el caso de los algoritmos de búsqueda, la idea es encontrar un dato específico dentro de una estructura como una lista. El más simple de todos es la búsqueda lineal, que va comparando uno por uno. Sirve siempre, pero si hay muchos datos puede volverse lenta. Se dice que su complejidad es O(n), lo que significa que el tiempo que tarda depende de cuántos elementos haya. Donde O(n) es el tiempo de ejecución que crece proporcionalmente al tamaño de los datos, por ejemplo, si tenemos una lista con 100 elementos, el algoritmo puede hacer hasta 100 pasos, si tuviera 1.000 haría 1.000 etc.

Después hay otra forma que es más rápida, pero tiene una condición: que los datos ya estén ordenados. Se llama búsqueda binaria, y lo que hace es dividir la lista a la mitad cada vez, descartando la parte que no sirve. Así encuentra el valor mucho más rápido, con complejidad es O (log n), aunque no se puede aplicar, en cualquier caso. Es necesario aclarar que O (log n) es el tiempo logarítmico que crece muy lentamente, aunque aumente la cantidad de datos. Es logarítmico porque el número de pasos crece como el logaritmo de n en base 2, como en el caso de la búsqueda binaria.

Justamente por eso ordenar se vuelve tan importante. Hay muchos algoritmos que hacen esto, y uno de los más usados es el quicksort. Fue inventado por Tony Hoare, tomando un problema de ordenamiento y lo descompone en subproblemas, que a su vez se descomponen en más subproblemas, trabajando eligiendo un valor pivote. Esto se logra mediante programación recursiva, donde la programación recursiva es una técnica que consiste en que una función se llame a si misma para resolver un problema hasta lograr el caso base. Luego separa los datos menores a un lado y los mayores al otro. Después repite el mismo proceso con esos dos grupos. Es como dividir el problema en partes más chicas.

Cabe destacar que quicksort como fue introducido por Tony inicialmente no fue perfecto, pero a medida que pasaron los años muchas personas colaboraron para ajustar y resolver las debilidades.

Algunas personas, ingenieros y programadores, han argumentado que cuando las particiones son lo suficientemente pequeñas, este tipo de ordenamiento resulta ser menos eficiente que usar otros metodos de ordenamiento.

Quicksort suele ser eficiente y rápido en la mayoría de los casos, aunque hay situaciones donde no rinde tanto. Igual, en general se comporta muy bien y es bastante usado en la práctica, porque además no necesita mucha memoria extra.

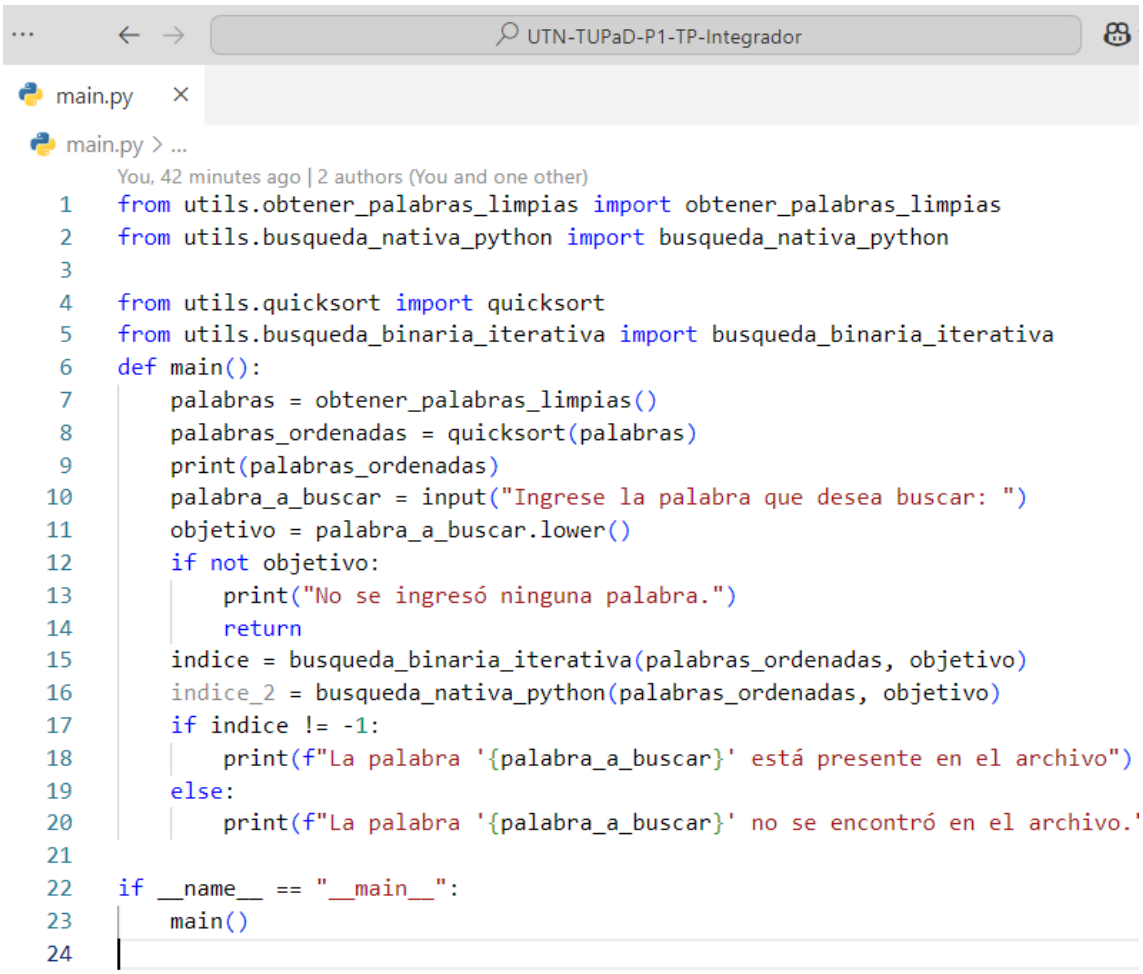
Anteriormente nombramos al tiempo, siempre refiriendonos al tiempo de ejecución del algoritmo, el peor escenario que podemos tener con referencia al tipo de ordenamiento quicksort es que el tiempo en relación al tamaño de la información, al tamaño de elementos de entrada que puede tener un algoritmo. Existe varios metodos para resolver o intentar resolver este problema, un metodo es usar siempre el mismo elemento en un conjunto, pero existe el riesgo de que el elemento elegido sea el primero o el último valor de un conjunto, otra posible solución seria la de elegir la media de un conjunto, pero esto agregaria mayor complejidad, ahora bien, una solución intermedia consistiria en elegir tres elementos del consjunto, por ejemplo, el primero, el del medio y el ultimo y usar el elemento del medio como pivote.

En este trabajo vamos a implementar una búsqueda simple y el quicksort usando Python, para ver en la práctica cómo funcionan estos conceptos.

Caso Práctico

Se desarrolló un programa en Python para buscar la existencia de una palabra en un texto. Se comparó el tiempo de búsqueda utilizando una búsqueda binaria y ordenamiento quicksort versus el método de búsqueda de arrays .index().

Programa principal:

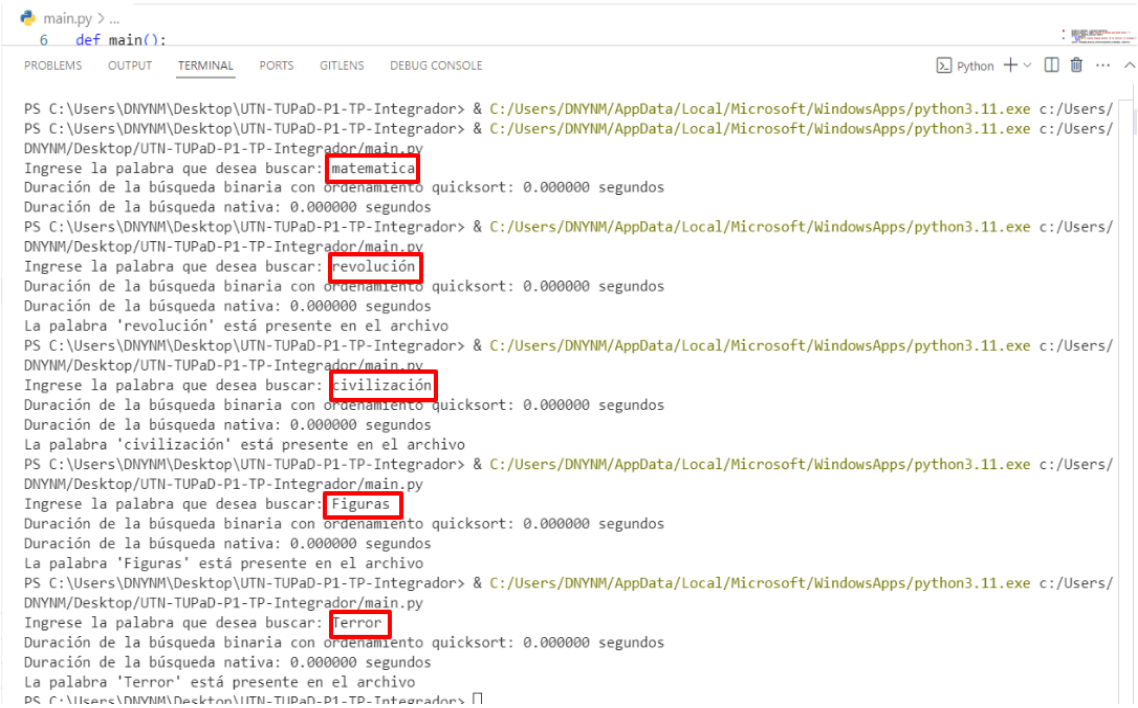


Metodología utilizada

- ChatGPT: arma un .txt de 5000 palabras al cual se le aplique la búsqueda
- Implementar lectura de archivos en Python
- Guardar las palabras en un array
- Ordenar el array de palabras con el ordenamiento quicksort
- Implementar una búsqueda binaria para determinar la existencia y posición de una palabra
- Implementar una búsqueda con el método de arrays .index()
- Implementar la funcionalidad de registrar los tiempos de ejecución
- Detectar y corregir fallas
- Ejecutar pruebas y registrar tiempos de ejecución

Resultados obtenidos

Captura de resultados:



Tanto para palabras no presentes (matemática) como para palabras presentes (revolución, civilización, Figuras, Terror) el tiempo de búsqueda es el mismo y es insignificante.

Conclusiones

Consideramos que la búsqueda binaria con el ordenamiento quick sort implementado sobre un archivo de texto de tamaño mediano a grande con el lenguaje Python es una implementación adecuada para resolver el problema debido a que la demora es tan escasa estando incluso por debajo del orden de magnitud de 10^{-6} lo que demuestra que es realmente óptimo sobre todo en estos tiempos donde la velocidad es muy importante y puede marcar la diferencia entre que un usuario utilice un programa o no.

Adicionalmente y con motivo de curiosidad, decidimos comparar los resultados de la elección hecha versus el método .index() de arrays para devolver la posición de la primera aparición del elemento, y nos llevamos la sorpresa de que también resultó ser muy eficiente realizando la búsqueda en la misma magnitud de tiempo.

Cada lenguaje de programación tiene sus ventajas, y los algoritmos son transversales a ellos, pero en Python tenemos la posibilidad de hacer una búsqueda eficiente simplemente utilizando un método.

Bibliografía

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introducción a los algoritmos* (Segunda edición) - Link
2. https://es.wikipedia.org/wiki/C._A._R._Hoare
3. Apuntes de la catedra programación 1. UTN-TUPaD-P1
4. Documentación de Python: https://www.python.org/doc/

Anexo

1. Repositorio: https://github.com/dante1704/UTN-TUPaD-P1-TP-Integrador
2. Link video: https://www.youtube.com/watch?v=SydI9f-8Ypg