Algoritmos de búsqueda y ordenamiento en Python.

- **Alumnos**:

  Agustin Nahuel Diaz Serafini – agudiaz962004@gmail.com

  Juan Ignacio Ceballo – juaneceba@gmail.com

- **Materia**: Programación I

- **Profesor/a**: Julieta Trapé

  Tutor: Miguel Barrera Oltra

- **Fecha de Entrega**: (9/6/2025)

Índice

1. Introducción :2
2. Marco Teórico: 3-8
3. Caso Práctico:
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

1. Introducción

Un algoritmo de búsqueda y ordenamiento no es solo un programa superficial. Se trata de procesos lógicos y pensamientos que permiten trascender a proyectos más grandes y complejos, sirviendo como una herramienta y un puente para complementar a muchas empresas actuales en la organización de sus actividades y proyectos.

En la programación, estos algoritmos pueden ser un fin en sí mismo, como un programa o una porción de código para lograr un orden de una lista o algo específico o una herramienta utilizada en un proceso más grande para lograr llevar adelante algoritmos más complejos, ayudando a organizar y filtrar datos e información que mejoren la eficiencia y el funcionamiento general de un programa.

Se propone entonces en este informe, lograr determinar un ejemplo simple en su función pero efectivo de estos algoritmos y dar una introducción a la profundidad que experimentan.

2. Marco Teórico.

Buscar y ordenar como herramienta de vida.

En un sentido general normalmente se piensa que un algoritmo de búsqueda y ordenamiento se basa en un programa que permite ordenar y filtrar datos. Si bien la función fundamental es únicamente esa, si se piensa en la definición general de un algoritmo (pasos específicos para lograr un fin específico) éstos están siendo utilizados siempre en nuestra vida cotidiana todo el tiempo. Por ejemplo, una maestra tiene que ordenar a los alumnos alfabéticamente, mirará todos los nombres y recorre esa lista mirando las primeras letras de cada nombre, al encontrar la primera A lo colocara al principio en una nueva lista hasta terminar con los nombres con A, luego mirará los que tienen letra B y así sucesivamente hasta la Z, en la lógica, está siguiendo un algoritmo de ordenamiento.

Buscar en programación.

En un ámbito de informática, los algoritmos de búsqueda son creados o utilizados

para buscar un dato o datos específicos de una lista o pieza de información, permitiendo una mayor eficiencia en el desarrollo de un proyecto más grande o para tener una mejor organización personal/ empresarial.

Dependiendo del objetivo y del tamaño de la lista hay muchos métodos para desarrollar un algoritmo de búsqueda:

Búsqueda Lineal.

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implican recorrer una lista de elementos uno por uno hasta encontrar un elemento específico. Se pueden realizar con un código simple y no muy largo, pero se torna ineficiente en listas con muchos datos.

Búsqueda Binaria.

En este algoritmo, se necesita que la lista en la que se aplica esté ordenada.
Se trata de mirar el elemento central de la lista, si ese es el elemento buscado entonces el objetivo se cumplió. Si no lo es, se compara el elemento central con el buscado, si el elemento buscado es mayor al obtenido entonces se aplicará el mismo proceso en la mitad superior de la lista, si no, se aplicará en la mitad inferior.

Para listas grandes, es mejor el método de búsqueda binaria pues es más eficiente que la secuencial. Sin embargo, la ventaja de la búsqueda lineal es que no depende que la lista en la que se trabaje esté ordenada, por lo que se debe tener en cuenta a la hora de elegir un método específico.

Ordenamiento en programación.

En la programación, ordenar datos es una herramienta clave para organizarlos y

poder trabajar con ellos de forma más eficiente.

Al igual que con la búsqueda, cuando se trata de ordenar una gran cantidad de información, el proceso puede requerir muchos recursos del sistema (tiempo de procesamiento, memoria, etc.). Por eso, la elección del método de ordenamiento no es algo menor. En listas pequeñas, a veces usar un algoritmo complejo puede ser más una complicación que una ventaja, ya que el tiempo que tarda en organizar la lista no se justifica por el tamaño reducido. Pero en cambio, si hablamos de grandes volúmenes de datos, lo ideal es aprovechar al máximo los métodos que mejoran el rendimiento, para lograr resultados más rápidos y eficientes.

Existen varios métodos de ordenamiento de datos:

Ordenamiento por burbuja (o bubble sort).

Este método funciona comparando elementos de a pares, uno al lado del otro. Si el elemento de la izquierda es mayor que el de la derecha, se intercambian de lugar. Luego se pasa al siguiente par, y así sucesivamente hasta llegar al final de la lista. Ese recorrido completo se repite varias veces, hasta que ya no hay más elementos desordenados.

Ordenamiento por selección.

Este algoritmo recorre toda la lista para encontrar el menor valor, lo intercambia con el que está en la primera posición, y luego repite el proceso con el resto de la lista, ignorando los que ya están ordenados al principio. Como si estuviera construyendo el orden final de un elemento por vez.

Ordenamiento por inserción (insertion sort).

Este algoritmo toma un elemento de la lista (a partir del segundo), y lo compara hacia atrás con los anteriores hasta encontrar la posición donde debe insertarse. Luego, repite el proceso con el siguiente elemento, y así sucesivamente hasta que toda la lista

queda ordenada.

Este método es eficiente en listas pequeñas o que ya están casi ordenadas. Sin embargo, al igual que los métodos anteriores, no es recomendable para grandes cantidades de datos porque el tiempo de procesamiento crece rápidamente.

Ordenamiento por mezcla (merge sort).

En este algoritmo se divide la lista en mitades de forma recursiva (es decir, repite el proceso dentro de sí mismo) hasta que cada sublista tenga solo un elemento (el cual ya está ordenado por definición). Luego, esas sublistas se van uniendo de dos en dos, ordenando al mezclarlas, hasta formar la lista original ya ordenada.

Este algoritmo es muy eficiente para trabajar con grandes cantidades de datos, porque mantiene un rendimiento estable incluso en los peores casos. Sin embargo, requiere más memoria, ya que se crean listas nuevas durante el proceso de mezcla.

Ordenamiento rápido (quicksort).

El algoritmo trabaja de la siguiente forma:

1. Se elige un elemento del conjunto de elementos a ordenar, al que se llama pivote.

2. Se resituan los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.

3. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

4. Se repite este proceso de forma recursiva para cada sublista mientras éstas

contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido:

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $O(n \cdot \log n)$.

- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.

- En el caso promedio, el orden es $O(n \cdot \log n)$.

La eficiencia del algoritmo Quicksort dependerá de la elección del pivote, ya que a partir de este se hace la partición como se mencionó anteriormente. Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección random siempre provoca que el algoritmo tenga un orden de $O(n^2)$ para ciertas permutaciones de los elementos en la lista. Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en $O(n)$ y asegura que hasta en el peor de los casos, el algoritmo sea $O(n \cdot \log n)$. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio. Con lo cual, la opción a medio camino es tomar tres elementos de la lista (usualmente el primero, el que ocupa justo la posición que divide la lista en dos, y el último) y

compararlos, eligiendo el valor del medio como pivote. A esto se le llama mediana de tres

Timsort

Timsort es un algoritmo moderno y muy eficiente que combina lo mejor de dos métodos clásicos: insertion sort (ordenamiento por inserción) y merge sort (ordenamiento por mezcla). Fue diseñado para funcionar muy bien en situaciones reales, donde muchas veces los datos ya están parcialmente ordenados.

La idea detrás de Timsort es bastante lógica y práctica. Primero, detecta automáticamente pequeñas partes de la lista que ya están ordenadas (a estas partes se las llama runs), y luego las ordena con inserción si son pequeñas, o las combina con mezcla si son más grandes. De esta forma, aprovecha el orden que ya existe para no hacer trabajo innecesario.

Para dar un ejemplo en python la función sorted() o .sort() usa Timsort, y es casi siempre la mejor opción práctica.

En general, los métodos simples como burbuja, selección o inserción son ideales para listas pequeñas, para fines educativos, o cuando no se necesita gran velocidad. Son fáciles de entender, implementar y funcionan bien en situaciones donde la eficiencia no es crítica. Además, su lógica se relaciona mucho con cómo solemos ordenar cosas en la vida cotidiana.

Por otro lado, algoritmos más potentes como merge sort, quick sort o timsort están pensados para trabajar con grandes volúmenes de datos. Estos métodos aprovechan estructuras más complejas y procesos recursivos que mejoran notablemente el rendimiento, especialmente en aplicaciones reales o cuando se procesan millones de elementos.

2. Caso Práctico

La situación que planteamos fue la de utilizar algoritmos de ordenamiento y búsqueda para un dato en una lista cuando no se sabe su posición en la misma, ni la extensión de dicha lista. El usuario ingresa el dato que desea buscar (en este caso un número), se simula el ingreso a la lista de éste en una posición random, también se define el length de la lista aleatoriamente, se simula la carga del resto de datos de forma aleatoria para llenar la lista. Posteriormente el programa define el algoritmo de ordenamiento adecuado a implementar según la extensión de la lista, se efectúa el ordenamiento, luego se utiliza un algoritmo de búsqueda binaria y se devuelve la posición del dato en la nueva lista ordenada.

Código Comentado

Programa principal:

```
import funciones as f

import random as r




num = f.valida_entero("Ingrese un número entero: ", "Por favor el número debe estar en el rango [-10000;10000] ", -10000, 10000)
```

```
        length = r.randint(1, 1000)



        # se generan numeros aleatorios y el guion bajo es una convención en Python que indica que no importa el valor de la variable del bucle.

        # el range es length - 1 para dejar un espacio para el numero ingresado por el usuario

        lista = [r.randint(-10000, 10000) for _ in range(length - 1)]



        # se crea el indice de forma aleatoria

        index = r.randint(0, length - 1)



        # se ingresa el valor del usuario en la posición aleatoria creada anteriormente

        lista.insert(index, num)




        # se analiza que algoritmo de ordenamiento utilizar según que tan grande sea el array

        if length <= 20:

            sorted_list = f.insertion_sort(lista)

        else:

            sorted_list = f.quicksort(lista)




        posicion = f.busqueda_binaria(num, sorted_list, length)



        if posicion == -1:
```

```
        print("No se ha encontrado el valor en la listaa")

    else:

        print(f"El número ingresado ({num}) se encuentra en la
posición {posicion} de la listaa")



    print(f"Length del array: {length-1}")

    for i in range(0, len(sorted_list)):

        print(f"key {i} ==> valor {sorted_list[i]}")
```

Archivo funciones.py

```
def mediana_de_tres(array):

    primero = array[0]

    medio = array[len(array) // 2] #valor de la posición media del
array (calcula con la división entera entre el lenght del array y 2)

    ultimo = array[-1]

    return sorted([primero, medio, ultimo])[1] #se ordena el array y
se utiliza el valor medio al acceder al indice 1



def quicksort(lista):



    if len(lista) <= 1:

        return lista



    pivot = mediana_de_tres(lista)

    # se utiliza el concepto de comprensión de listaas para crear
listaas con condiciones en una sola linea

    mayores = [x for x in lista if x > pivot]
```

```
    menores = [i for i in lista if i < pivot]

    medio = [y for y in lista if y == pivot]

    return quicksort(menores) + medio + quicksort(mayores)



def valida_entero(mensaje, error, min = float("-Inf"), max = float("Inf")):

    num = int(input(f"{mensaje}"))

    while num < min or num > max:

        num = int(input(f"{error}, {mensaje}"))

    return num




def insertion_sort(array):

    for i in range(1, len(array)):

        actual = array[i]

        j = i - 1


        # Mover hacia la derecha los elementos mayores que el actual

        while j >= 0 and array[j] > actual:

            array[j + 1] = array[j]

            j -= 1


        # Insertar el actual en la posición correcta

        array[j + 1] = actual

    return array
```

```
def busqueda_binaria(num, sorted_lista, length):



    izquierda = 0

    derecha = len(sorted_lista) - 1



    while izquierda <= derecha:

        medio = (izquierda + derecha) // 2

        valor_medio = sorted_lista[medio]



        if valor_medio == num:

            return medio  # Devuelve el índice donde lo encontró

        elif num < valor_medio:

            # descarta los valores más grandes y el limite derecho
empieza en medio - 1 (para descartar tambien el valor de medio)

            derecha = medio - 1

        else:

            # descarta los valores mas chicos y el limite izquierdo
empieza en medio + 1 (para descartar también el valor de medio)

            izquierda = medio + 1



    return -1  # No encontrado
```

En cuanto a la elección del método de ordenamiento, decidimos implementar un umbral para que el programa decida elegir cuál método utilizar entre quicksort y insertion sort. Dicho umbral es la extensión o length del array, ya que según la investigación que utilizamos, para listas chicas, insertion sort es más eficiente que otros, y en el caso de listas más grandes quicksort es el más eficiente. Según lo que investigamos, el length que

indica que una lista es grande o chica es aproximadamente 20, con lo cual decidimos que ese número sea nuestro umbral.

---

3. Metodología Utilizada

- Debimos primero investigar bien sobre los algoritmos tanto de búsqueda como de ordenamiento y su funcionamiento. Y también cual era más óptimo para cada caso.
- Utilizamos Visual Studio Code como IDE. En cuanto a las librerías, utilizamos el módulo random propio de Python. Para el repositorio usamos Github y para el versionado de código Git .

---

4. Resultados Obtenidos

Con las pruebas iniciales nos dimos cuenta que ambos algoritmos, tanto el de búsqueda como el de ordenamiento funcionaban pero al utilizar un for para recorrer la posición del dato en la lista y devolvérsela al usuario, nos encontramos con un error. Ya que estábamos iterando sobre la lista original y la búsqueda se aplicó a la lista ordenada. Con lo cual, tuvimos que corregir el código e iterar sobre la lista ordenada y a partir de ello, las pruebas dieron los resultados que esperábamos. Quitando esa situación, el resto operó sin problemas.

5. Conclusiones

Aprendimos a utilizar algunos de los algoritmos de ordenamiento y a implementarlos según el contexto en el que trabajemos, en este caso la longitud de las listas con las que operamos. Además, pudimos hacer uso práctico de funciones recursivas, un recurso que es muy utilizado en programación.

Este trabajo fue especialmente útil para ayudarnos a decidir en qué momento usar un método u otro según convenga, aportandonos algo de experiencia a la hora de desarrollar código más eficiente, lo que puede contribuir en futuros proyectos.

Hablando del código en sí, es una buena fuente de optimización hacia futuros códigos que desarrollemos que necesiten acciones de búsqueda y ordenamiento automático.

6. Bibliografía

Wikipedia contributors. (s.f.). *Quicksort*. Wikipedia. Recuperado el 9 de junio de 2025, de https://es.wikipedia.org/wiki/Quicksort

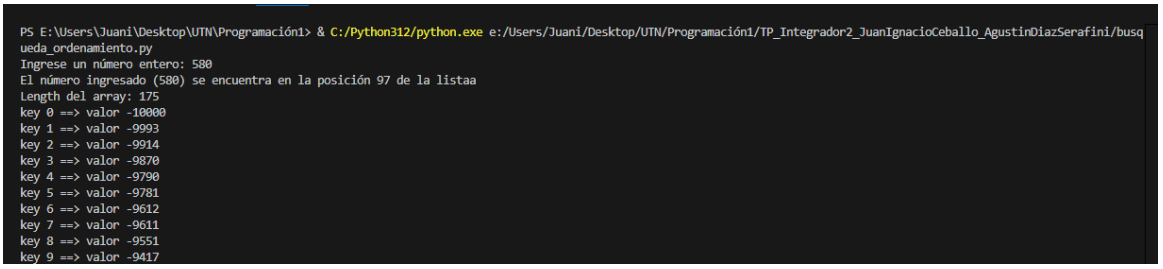
Miller, B. (s.f.). *Sorting and Searching*. Runestone Interactive. Recuperado el 9 de junio de 2025, de https://runestone.academy/ns/books/published/pythoned/SortSearch/toctree.html

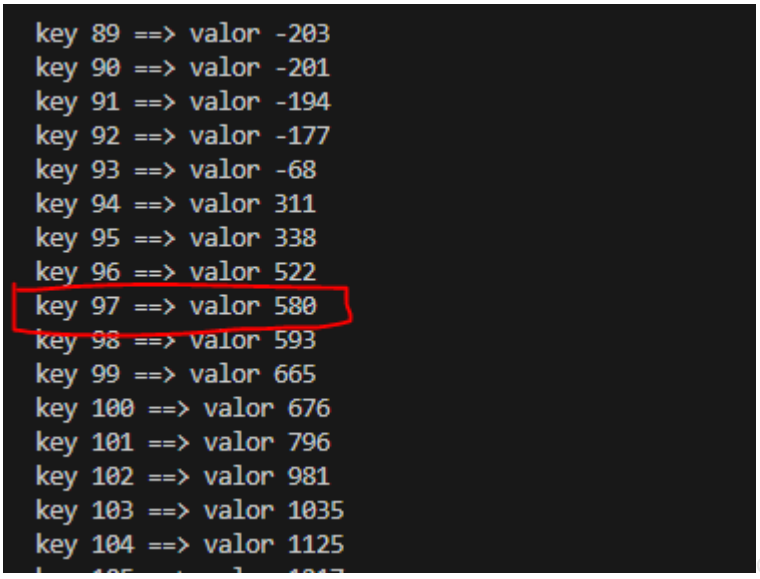
Baeldung. (s.f.). *Timsort algorithm in computer science*. Baeldung. Recuperado el 9 de junio de 2025, de https://www-baeldung-com.translate.goog/cs/timsort?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Skerritt, D. (2019, agosto 6). *Timsort: A very fast, hybrid sorting algorithm in Python*. Skerritt.blog. https://skerritt.blog/timsort/

7. Anexo

MUESTRA DE FUNCIONAMIENTO.





v.† Recomendaciones para la presentación

- Formato del archivo: **.docx** o **.pdf**
- Tipografía: Arial o Calibri, tamaño 11 o 12
- Interlineado: 1,5
- Márgenes estándar (2.5 cm)
- Portada opcional con el logo de la institución (si se requiere)

- Entrega digital mediante plataforma institucional o por correo