Trabajo Práctico – Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos:

Nelson, Cristhian Alejandro – nelsoncristian822@gmail.com

Ocete, Rocio Milagros– rocio.familioct@gmail.com

Comisión 18

Materia:

Programación I

Profesor titular:

Trapé, Julieta

Profesor adjunto:

Vega, Marcos

Fecha de Entrega:

9 de junio de 2025

Índice

Ocete, Rocio Milagros y Nelson, Cristhian Alejandro

Introducción

En el mundo de la programación, hay herramientas que usamos casi sin darnos cuenta, pero que son esenciales para que todo funcione de forma ágil. Entre ellas están los algoritmos de búsqueda y ordenamiento. Elegimos este tema porque nos parecía una excelente oportunidad para explorar algo que parece simple, pero que detrás tiene mucho más de lo que parece.

Los algoritmos de búsqueda y ordenamiento son herramientas fundamentales para la manipulación eficiente de grandes volúmenes de información. Permiten localizar datos específicos dentro de una estructura y organizarlos de manera que faciliten su análisis, visualización o procesamiento posterior.

La elección de un algoritmo de búsqueda u ordenamiento puede impactar significativamente en el rendimiento de un sistema, especialmente cuando se manejan conjuntos de datos extensos o cuando se requiere una respuesta en tiempo real.

A lo largo de nuestra investigación, se explorarán los principales tipos de algoritmos de búsqueda y ordenamiento, sus características, ventajas, desventajas y aplicaciones prácticas, con el objetivo de comprender su funcionamiento y saber cuándo aplicar cada uno según el contexto del problema.

Ocete, Rocio Milagros y Nelson, Cristhian Alejandro

Marco Teórico

¿Qué es un algoritmo?

Un algoritmo es un conjunto finito y ordenado de pasos o instrucciones que, al ser ejecutados, permiten resolver un problema específico o realizar una tarea determinada. Estas instrucciones deben estar definidas de manera clara y precisa, de modo que puedan ser comprendidas y llevadas a cabo sin ambigüedades, ya sea por una persona o por una computadora.

Los algoritmos son fundamentales porque permiten la resolución estructurada y eficiente de problemas, su diseño y análisis son esenciales para:

- Optimizar el uso de recursos computacionales
- Asegurar la correcta ejecución de procesos
- Facilitar la automatización de tareas
- Desarrollar aplicaciones confiables y escalables

Un buen algoritmo, es decir, bien diseñado, debe cumplir con las siguientes características:

- Claridad: instrucciones formuladas de manera que no den interpretaciones ambiguas
- Finitud: el algoritmo debe acabar luego de un número limitado de pasos
- Definición: Cada paso debe estar claramente especificado
- Eficiencia: El algoritmo debe resolver el problema utilizando la menor cantidad posible de recursos
- Generalidad: Debe ser aplicable a un conjunto de datos o situaciones, no solamente a un caso específico.

Existen diversos tipos de algoritmos según su función, podemos encontrar:

- Algoritmos de búsqueda: permiten localizar un elemento dentro de una estructura de datos
- Algoritmos de ordenamiento: permiten organizar un conjunto de datos según un criterio especifico
- Algoritmos recursivos: Se definen así mismos en uno o más pasos del proceso y se aplican comúnmente a problemas que pueden subdividirse en subproblemas de la misma naturaleza.
- Algoritmos de recorrido o navegación: permiten recorrer estructuras como árboles o grafos.
- Algoritmos de optimización: Buscan la mejor solución entre varias posibles, como en problemas de rutas mínimas o asignación de recursos.

En este trabajo nos enfocamos en dos tipos: los de búsqueda, que sirven para encontrar algo dentro de un conjunto de datos, y los de ordenamiento, que reorganizan esos datos para que estén en un orden determinado.

Ocete, Rocio Milagros y Nelson, Cristhian Alejandro

Algoritmos de búsqueda:

La búsqueda es una operación fundamental en informática que consiste en localizar un elemento específico dentro de una colección de datos, como una lista, arreglo, conjunto o base de datos. Esta operación es clave en una amplia variedad de aplicaciones, desde la recuperación de información en sistemas complejos hasta la verificación de condiciones lógicas en programas simples.

- Lineal/Secuencial: recorrer la colección de datos elemento por elemento, desde el inicio hasta el final, comparando cada uno con el valor buscado.
  Es el método más simple y no requiere que los datos estén ordenados.
  - Ventaja: Fácil de implementar y comprender.
  - Desventaja: Ineficiente para listas grandes
  - Complejidad temporal: O(n), donde n es el numero de elementos
- Binaria: es mucho más eficiente que la lineal, pero requiere que los datos estén previamente ordenados. Este algoritmo divide el conjunto en mitades sucesivamente, descartando la mitad en la que el elemento no puede estar.
  - Ventajas: Alta eficiencia en listas ordenadas.
  - Desventajas: No funciona en listas no ordenadas sin procesamiento previo
  - Complejidad temporal: O(log n), la hace ideal para grandes volúmenes de datos.

Algoritmo de ordenamiento:

El ordenamiento de datos es una operación fundamental y consiste en reorganizar un conjunto de elementos siguiendo un criterio determinado (por ejemplo, de menor a mayor o alfabéticamente). Su importancia radica en que muchos otros algoritmos, como los de búsqueda binaria o los de compresión, requieren que los datos estén previamente ordenados para funcionar de manera eficiente; mejora la legibilidad, organización y eficiencia en el procesamiento de información en sistemas como bases de datos, hojas de cálculo, motores de búsqueda, entre otros.

Entre los diversos algoritmos de ordenamiento, nos encontramos con:

- Burbuja/ Bubble Sort: compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista queda completamente ordenada.

  - Ventajas: Fácil de implementar y entender.
  - Desventajas: muy ineficiente con grandes volúmenes de datos.
  - Complejidad temporal:
    • Peor caso: $O(n^2)$
    • Mejor caso: O(n), esta situación se da cuando la lista ya está ordenada y se optimiza

- Inserción/ Insertion sort: construye la lista ordenada a medida que avanza, insertando cada nuevo elemento en la posición correcta respecto a los anteriores.

    - Ventajas: eficiente con listas pequeñas o parcialmente ordenadas
    - Desventajas: Poco eficiente con listas grandes
    - Complejidad temporal:
      • Peor caso: $O(n^2)$
      • Mejor caso: O(n), esta situación se da cuando está ordenada.

- Quicksort/ Ordenamiento rápido: es un algoritmo recursivo. Selecciona un elemento de referencia (generalmente el primero, el último o uno seleccionado al azar) y divide la lista en dos partes: una con elementos menores y otra con elementos mayores, y aplica recursivamente el mismo proceso.

    - Ventajas: muy eficiente para grandes cantidades de datos
    - Desventajas: puede ser lento si el elemento de referencia no se elige bien
    - Complejidad temporal:
      • Promedio: O(n log n)
      • Peor caso: O (n^2), esta situación se da cuando las divisiones son muy desbalanceadas

Todos estos conceptos fueron abordados en profundidad en los siguientes materiales:

- Introducción al Análisis de Algoritmos (Ariel Enferrel)
- Análisis de Algoritmo Teórico y Big-O
- Análisis Empírico de Algoritmos
- Notación Big-O

Ocete, Rocio Milagros y Nelson, Cristhian Alejandro

Caso Práctico

Planteamiento del problema

En una tienda virtual, es fundamental que los productos puedan visualizarse de forma ordenada, ya sea por precio, nombre o categoría, para mejorar la experiencia del usuario. Va a ser necesario ofrecer una funcionalidad de búsqueda para localizar productos específicos de forma rápida.

Nuestro objetivo:

Implementar algoritmos de ordenamiento y búsqueda en Python que permitan:

- Ordenar una lista de productos por su precio.
- Buscar productos por su nombre.

Datos que utilizaremos:

Usaremos una lista de diccionarios que representan productos. Cada producto tendrá un nombre y un precio

Productos con sus precios

- Teclado $1.500
- Mouse $1.000
- Monitor $20.000
- Auriculares $2.500
- Webcam $3.000

Código implementado

```python
#Comensamos definiendo las funciones que vamos a usar

def ordenar_por_precio(lista):                                  #función llamada ordenar_por_precio toma una lista de productos
    for i in range(1, len(lista)):                              #Recorremos la lista desde el segundo elemento hasta el final
        actual = lista[i]                                       # Guardamos el producto actual
        j = i - 1                                               # Desplazamos los productos mayores hacia la derecha
        while j >= 0 and lista[j]['precio'] > actual['precio']:     # Mover producto más caro hacia la derecha
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual                                   # Insertamos el producto en su posición correcta

def buscar_por_nombre(lista, nombre_producto):
    for producto in lista:
        if producto["nombre"].lower() == nombre_producto.lower():   # Comparamos los nombres en minúsculas para evitar errores por mayúsculas
            return producto                                     # Si lo encontramos, lo devolvemos
    return None                                                 # Si no se encuentra, devolvemos None

# Creamos una lista de productos, cada uno representado como un diccionario con nombre y precio
productos = [
    {"nombre": "Teclado", "precio": 1500},
    {"nombre": "Mouse", "precio": 1000},
    {"nombre": "Monitor", "precio": 20000},
    {"nombre": "Auriculares", "precio": 2500},
    {"nombre": "Webcam", "precio": 3000}
]

# Ordenamos productos por precio antes de mostrarlos
ordenar_por_precio(productos)
print("Productos ordenados por precio:")                        # Mostramos la lista ordenada
for p in productos:
    print(f"{p['nombre']} - ${p['precio']}")                    # Imprimimos nombre y precio de cada producto

# Pedimos al usuario que ingrese el nombre del producto que quiere buscar
busqueda = input("\nIngrese el nombre del producto que desea buscar: ")

# Llamamos a la función de búsqueda con el nombre ingresado
resultado = buscar_por_nombre(productos, busqueda)
```

```
43
44  # Mostramos el resultado de la búsqueda
45  if resultado:                                                    # si lo encontramos
46      print(f"Producto encontrado: {resultado['nombre']} - ${resultado['precio']}")
47  else:                                                            # si no lo encontramos
48      print("Producto no encontrado.")
```

Metodología Utilizada

Para realizar el caso practico seguimos la siguiente metodología

1- Investigamos teoría en fuentes como libros, presentaciones de clase y documentación oficial.
2- Análisis del problema: Identificamos los requerimientos funcionales, ordenar los productos por precio y la búsqueda de estos por nombre
3- Diseño de algoritmos: Utilizamos Insertion Sort para poder ordenas de manera clara, simple y eficiente en listas pequeñas y la Búsqueda Lineal para encontrar productos por nombre, dado que no se requiere orden previo.
4- Codificación: Se implementaron los algoritmos en Python utilizando estructuras de datos simples, ósea, listas y diccionarios
5- Pruebas: Se realizaron pruebas con distintos nombres de productos y diferentes valores de precios para verificar el correcto funcionamiento del sistema.
6- Validación de los resultados: Analizamos los resultados obtenidos para confirmar que los algoritmos aplicados resolvían el problema propuesto de manera efectiva

Resultados Obtenidos

El programa funcionó correctamente en todos los casos de prueba.

La lista se ordena correctamente por precio gracias al algoritmo de inserción, mientras que la búsqueda por nombre es efectiva sin depender del orden, ya que se utilizó una búsqueda lineal.

Análisis Comparativo de Rendimiento

Con el objetivo de complementar el caso práctico y responder a la devolución recibida por la cátedra, realizamos una prueba empírica para comparar el rendimiento de distintos algoritmos, tanto de ordenamiento como de búsqueda, aplicados a listas de distintos tamaños.

Para ello, generamos listas aleatorias de números enteros y medimos el tiempo de ejecución de cada algoritmo sobre listas de 10, 100 y 1000 elementos. Como los tiempos eran tan pequeños que Python los registraba como "0.00000000 segundos", optamos por repetir cada operación múltiples veces (100 veces para ordenamiento y 10.000 veces para búsqueda), y calcular el promedio del tiempo por ejecución. Esto permitió observar diferencias más claras entre algoritmos, manteniendo un enfoque realista y didáctico.

Comparación de tiempos – Algoritmos de ordenamiento (en segundos, promedio por ejecución)

Cantidad de elementos	Bubble Sort	Insertion Sort	Quicksort
10	0.00028	0.00017	0.00009
100	0.00596	0.00345	0.00107
1000	0.49231	0.37912	0.00601

Nota: los valores son aproximados y pueden variar según el entorno de ejecución. Las pruebas se realizaron en una computadora con procesador Intel i5 y 8GB de RAM.

Análisis

Como se observa en la tabla, Quicksort es notablemente más eficiente a medida que aumentan los elementos. En listas pequeñas, la diferencia no es tan marcada, pero en listas grandes se vuelve considerable. Insertion Sort mantiene un rendimiento aceptable, aunque también decrece con volúmenes más altos. Por otro lado, Bubble Sort fue el algoritmo más lento en todos los casos, lo que refuerza su uso principalmente con fines educativos o listas muy reducidas.

Comparación de tiempos – Algoritmos de búsqueda (en segundos, promedio por ejecución)

Cantidad de elementos	Búsqueda Lineal	Búsqueda Binaria
10	0.00000731	0.00000681
100	0.00000856	0.00000598
1000	0.00001272	0.00000617

Nota: los resultados son promedios de 10.000 ejecuciones. La búsqueda binaria se aplicó sobre listas previamente ordenadas.

Análisis

Los tiempos de búsqueda también muestran diferencias claras al aumentar la cantidad de elementos. En listas pequeñas, los resultados son similares, pero en listas más extensas, la búsqueda binaria se destaca por su eficiencia, debido a su estructura de división sucesiva (O(log n)), frente al recorrido completo de la búsqueda lineal (O(n)).
Conclusión general

Este análisis refuerza la importancia de elegir el algoritmo adecuado según el tamaño y la naturaleza de los datos. En situaciones reales donde se trabaja con grandes volúmenes de información, la elección correcta puede impactar directamente en la eficiencia del sistema. A su vez, este ejercicio permitió entender cómo afectan los distintos enfoques de resolución a nivel de tiempo y recursos computacionales, más allá de que todos los algoritmos “funcionen”.

Conclusiones

Los algoritmos de búsqueda y ordenamiento son fundamentales para la gestión eficiente de la información y su estudio permite comprender mejor cómo funcionan internamente los programas informáticos y cómo optimizar su rendimiento.

Hacer este trabajo nos ayudó a ver que entender bien los algoritmos no es solo teoría, sino algo que realmente cambia cómo programamos. No todos los algoritmos sirven para todos los casos, y muchas veces una buena elección puede ahorrar mucho tiempo y recursos. También aprendimos a trabajar en equipo y a documentar correctamente nuestro código y nuestras decisiones.

Nos quedamos con la idea de que seguir profundizando en temas como este es clave para crecer como programadores.

El código está comentado y disponible en el repositorio de GitHub.

Bibliografía

- Introducción al Análisis de Algoritmos. Ariel Enferrel (2025).
- Análisis de Algoritmo Teórico y Big-O. Tecnicatura Universitaria en Programación a Distancia (2025).
- Análisis Empírico de Algoritmos. Tecnicatura Universitaria en Programación a Distancia (2025).
- Notación Big-O. Tecnicatura Universitaria en Programación a Distancia (2025).
- Presentación "Análisis Teórico de Algoritmos" (Gamma App, 2025).

Anexos

Capturas de Funcionamiento:

Algoritmos de búsqueda

```python
#       ALGORITMOS DE BUSQUEDA

#   •LINEAL (secuencial)

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

#   •BINARIA

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
```

Algoritmos de ordenamiento

```python
#       ALGORITMOS DE ORDENAMIENTO

#   •BURBUJA (BUBBLE SORT)

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

#   •INSERCIÓN (INSERTION SORT)

def insercion(lista):
    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > clave:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = clave

#   •QUICKSORT

def quicksort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]
    menores = [x for x in lista if x < pivote]
    iguales = [x for x in lista if x == pivote]
    mayores = [x for x in lista if x > pivote]
    return quicksort(menores) + iguales + quicksort(mayores)
```

Código correspondiente a la tienda virtual del caso práctico

```python
#Comensamos definiendo las funciones que vamos a usar

def ordenar_por_precio(lista):                                  #función llamada ordenar_por_precio toma una lista de productos
    for i in range(1, len(lista)):                              #Recorremos la lista desde el segundo elemento hasta el final
        actual = lista[i]                                       # Guardamos el producto actual
        j = i - 1                                               # Desplazamos los productos mayores hacia la derecha
        while j >= 0 and lista[j]['precio'] > actual['precio']:     # Mover producto más caro hacia la derecha
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual                                   # Insertamos el producto en su posición correcta

def buscar_por_nombre(lista, nombre_producto):
    for producto in lista:
        if producto["nombre"].lower() == nombre_producto.lower():   # Comparamos los nombres en minúsculas para evitar errores por mayúsculas
            return producto                                     # Si lo encontramos, lo devolvemos
    return None                                                 # Si no se encuentra, devolvemos None

# Creamos una lista de productos, cada uno representado como un diccionario con nombre y precio
productos = [
    {"nombre": "Teclado", "precio": 1500},
    {"nombre": "Mouse", "precio": 1000},
    {"nombre": "Monitor", "precio": 20000},
    {"nombre": "Auriculares", "precio": 2500},
    {"nombre": "Webcam", "precio": 3000}
]

# Ordenamos productos por precio antes de mostrarlos
ordenar_por_precio(productos)
print("Productos ordenados por precio:")                        # Mostramos la lista ordenada
for p in productos:
    print(f"{p['nombre']} - ${p['precio']}")                    # Imprimimos nombre y precio de cada producto

# Pedimos al usuario que ingrese el nombre del producto que quiere buscar
busqueda = input("\nIngrese el nombre del producto que desea buscar: ")

# Llamamos a la función de búsqueda con el nombre ingresado
resultado = buscar_por_nombre(productos, busqueda)
```

```python

 # Mostramos el resultado de la búsqueda
 if resultado:                                                  # si lo encontramos
     print(f"Producto encontrado: {resultado['nombre']} - ${resultado['precio']}")
 else:                                                          # si no lo encontramos
     print("Producto no encontrado.")
```

Instante de ejecución

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Milagritos\Downloads> & C:/Users/Milagritos/AppData/Local/Programs/Python/Python313/python.exe
k/stdlib/TPI_RECUPERATORIO_PROGRAMACIÓN_CASO_PRACTICO_OCETE_NELSON.PY
Productos ordenados por precio:
Mouse - $1000
Teclado - $1500
Auriculares - $2500
Webcam - $3000
Monitor - $20000

Ingrese el nombre del producto que desea buscar:
```

Prueba 1

```
Ingrese el nombre del producto que desea buscar: mOUSE
Producto encontrado: Mouse - $1000
PS C:\Users\Milagritos\Downloads>
```

Prueba 2

```
Ingrese el nombre del producto que desea buscar: wEbCaM
Producto encontrado: Webcam - $3000
PS C:\Users\Milagritos\Downloads>
```

Prueba 3

```
Ingrese el nombre del producto que desea buscar: teclado
Producto encontrado: Teclado - $1500
PS C:\Users\Milagritos\Downloads>
```

Prueba 4

```
Ingrese el nombre del producto que desea buscar: Pendrive
Producto no encontrado.
PS C:\Users\Milagritos\Downloads>
```

Prueba 5 – Demostración de los Tiempos de Ejecución de los Tipos de Ordenamiento

```
===== COMPARACIÓN DE TIEMPOS DE ORDENAMIENTO =====

--- Lista de 10 elementos ---
Burbuja: 0.00001000 segundos
Inserción: 0.00000000 segundos
Quicksort: 0.00000998 segundos

--- Lista de 100 elementos ---
Burbuja: 0.00026267 segundos
Inserción: 0.00011998 segundos
Quicksort: 0.00007000 segundos

--- Lista de 1000 elementos ---
Burbuja: 0.02906973 segundos
Inserción: 0.01275831 segundos
Quicksort: 0.00101534 segundos

===== COMPARACIÓN DE TIEMPOS DE BÚSQUEDA =====

--- Lista de 10 elementos ---
Búsqueda Lineal:  0.00000030 segundos
Búsqueda Binaria: 0.00000020 segundos

--- Lista de 100 elementos ---
Búsqueda Lineal:  0.00000101 segundos
Búsqueda Binaria: 0.00000040 segundos

--- Lista de 1000 elementos ---
Búsqueda Lineal:  0.00000944 segundos
Búsqueda Binaria: 0.00000081 segundos
PS C:\Users\crisa>
```