

Datos Generales

- **Título del trabajo:** La Importancia De búsqueda y Ordenamiento Para El Programador
- **Alumnos:** Nazareno Romero – nazareno.romero2910@gmail.com

Leandro Torres Galarzo – leandrogalarzo@gmail.com

- **Materia:** Programación I
- **Profesor/a:** (Cinthia Rigoni)
- **Fecha de Entrega:** 09/6/2025

Índice

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

1. Introducción

El trabajo tiene como objetivo abordar el estudio de los **algoritmos de búsqueda y ordenamiento**, dos conceptos fundamentales en el área de programación. La elección de este tema se basa en su importancia para resolver problemas de manera eficiente y en su presencia constante en múltiples aplicaciones informáticas.

En programación, buscar y ordenar datos son tareas esenciales. Permiten encontrar información específica, organizarla y mejorar el rendimiento de programas que manejan grandes volúmenes de datos, como bases de datos, sistemas de archivos o plataformas de análisis.

Lo demostraremos en este trabajo mediante la teoría y un caso práctico en Python.

2. Marco Teórico

BÚSQUEDA: Los algoritmos de búsqueda son como "detectives digitales" que nos ayudan a encontrar información específica dentro de grandes conjuntos de datos. Utilizan reglas y procesos sistemáticos para reducir la cantidad de comparaciones necesarias y encontrar rápidamente lo que buscamos.

La búsqueda es una operación clave en programación y estructuras de datos, ya que permite localizar elementos de forma eficiente, especialmente cuando se trabaja con mucha información.

Esta operación es fundamental en áreas como:

- **Bases de datos:** para buscar registros específicos.
- **Motores de búsqueda:** como Google, para encontrar resultados relevantes.
- **Inteligencia artificial:** para identificar patrones y tomar decisiones.
- **Análisis de datos:** para extraer información útil de grandes volúmenes de datos.

Tipos de Algoritmos de Búsqueda

• Búsqueda lineal

También llamada búsqueda secuencial. Este algoritmo recorre cada elemento de una lista o matriz y lo compara con el valor buscado. Si encuentra una coincidencia, retorna su posición; si llega al final sin encontrarlo, se asume que el elemento no está presente.

Pros:

- Sencilla y fácil de implementar.
- Funciona con listas sin ordenar.
- Ideal para conjuntos de datos pequeños.

Contras:

- No es eficiente para listas grandes.
- Su tiempo de ejecución es $O(n)$ en el peor caso.

¿Cuándo usarla?

- Cuando los datos no están ordenados o la lista es pequeña

• **Búsqueda binaria**

Es un algoritmo eficiente para listas ordenadas. Divide la lista por la mitad y compara el valor buscado con el elemento del medio. Si no lo encuentra, continúa en la mitad correspondiente hasta dar con el valor o agotar las opciones.

Pros:

- Muy eficaz en listas grandes y ordenadas.
- Más rápida que la búsqueda lineal.
- De implementación relativamente sencilla.

Contras:

- Solo funciona con listas ordenadas.
- Requiere acceso aleatorio a los elementos, lo que puede ser una desventaja en algunas estructuras.

¿Cuándo usarla?

- Cuando los datos están ordenados y se necesita eficiencia en listas grandes.

• **Otras**

Búsqueda por interpolación

- Estima la posición del elemento buscado en listas numéricas ordenadas.
- Funciona bien si los datos están distribuidos de forma uniforme.

Hashing (búsqueda por función hash)

- Asigna una clave única a cada elemento mediante una función hash.
- Permite acceder directamente al dato en tiempo constante promedio ($O(1)$).

Búsqueda por comparación

- Usa criterios para descartar múltiples elementos a la vez.
- Se aplica sobre estructuras ordenadas y busca reducir el número de comparaciones

Ordenamiento: Un algoritmo de ordenamiento es un conjunto de instrucciones que recibe una lista o arreglo como entrada y organiza sus elementos en un orden específico. Generalmente, este orden puede ser ascendente (A-Z, 0-9) o descendente (Z-A, 9-0), y puede aplicarse tanto a números como a letras.

¿Por qué son importantes los algoritmos de ordenamiento?

- Los algoritmos de ordenamiento son fundamentales en informática porque permiten organizar datos para facilitar otras operaciones, como la **búsqueda**, el análisis o la manipulación de datos. También son la base de técnicas como divide y vencerás, estructuras de datos eficientes y algoritmos en bases de datos. Un buen ordenamiento puede reducir significativamente la complejidad de los problemas.

Ordenamientos más comunes

- **1. Ordenamiento por burbuja (Bubble Sort)**
Compara pares de elementos adyacentes e intercambia sus posiciones si están en el orden incorrecto. Este proceso se repite hasta que toda la lista esté ordenada.
Peor caso: $O(n^2)$ – muy ineficiente para listas grandes.
- **2. Ordenamiento por selección (Selection Sort)**
Asume que el primer elemento es el menor y busca en el resto de la lista uno menor. Si lo encuentra, intercambia. Repite este proceso para cada posición.
Peor caso: $O(n^2)$ – poco eficiente en grandes volúmenes de datos.
- **3. Ordenamiento por inserción (Insertion Sort)**
Toma cada elemento y lo coloca en la posición correcta respecto a los anteriores, como si se “insertara” ordenadamente en una lista ya ordenada.
Peor caso: $O(n^2)$ – aunque es eficiente para listas pequeñas o casi ordenadas.
- **4. Ordenamiento rápido (Quick Sort)**
Usa la técnica "divide y vencerás". Elige un elemento pivote y divide la lista en dos partes: una con elementos menores y otra con mayores. Ordena recursivamente.
Peor caso: $O(n^2)$ (cuando los datos están muy desordenados o iguales).
Mejor caso/promedio: $O(n \log n)$ – muy eficiente en la mayoría de los casos.
- **5. Ordenamiento por mezcla (Merge Sort)**
También aplica "divide y vencerás". Divide el arreglo en sublistas, las ordena y luego las fusiona.
Complejidad en todos los casos: $O(n \log n)$ – rendimiento muy estable y predecible.

3. Caso Práctico

- Creamos un código en Python para mostrar distintos casos de búsqueda y ordenamiento.
 - Bubble sort, insertion sort, selection sort y búsqueda binaria fueron los casos que hicimos para que se muestren en la terminal
-

4. Metodología Utilizada

- Comunicación via WhatsApp y Discord.
 - Investigación: búsqueda por navegador en sitios web.
 - Resumimos toda la información para adaptarla al documento y las diapositivas del Canva creado.
 - Diseño del código en el Visual Studio Code.
 - Separamos las consignas para que sean explicadas en el video, grabamos y editamos en Capcut.
 - Uso de Git Hub para subir el trabajo
-

5. Resultados Obtenidos

Bubble sort, insertion sort, selection sort, búsqueda binaria ordeno todo correctamente

6. Conclusiones

Los algoritmos de ordenamiento y búsqueda son pilares fundamentales en el desarrollo de software, ya que permiten organizar y acceder a los datos de manera eficiente. A lo largo del trabajo se analizaron diferentes tipos de algoritmos, se implementaron en código y se evaluaron sus características, ventajas y limitaciones.

Cada algoritmo presenta comportamientos distintos según el contexto. En conclusión, la correcta elección e implementación de estos algoritmos no solo mejora el rendimiento de un programa, sino que también permite resolver problemas de forma más precisa y escalable. Este conocimiento es una herramienta clave en la formación de cualquier programador

7. Bibliografía

- SITIOS WEB: LUIGI'S BOXS, [geeksforgeeks.org](https://www.geeksforgeeks.org/), [FREECODECAMP](https://www.freecodecamp.org/) y [BUILTIN](https://www.builtwithpython.com/)
 - Material dado por la universidad, videos, powerpoint
-

8. Anexos

```
1 # Este programa permite realizar operaciones con conjuntos de dígitos de DNIs y años de nacimiento.
2 import datetime
3
4 def es_bisiesto(año):
5     return (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0)
6
7 def operaciones_con_dnis():
8     print("\n--- OPERACIONES CON DNIs ---")
9     dnis = input("Ingrese los DNIs separados por coma: ").split(",")
10    dnis = [dni.strip() for dni in dnis]
11
12    conjuntos = [set(int(d) for d in dni) for dni in dnis]
13    print("\nConjuntos de dígitos únicos:")
14    for i, conjunto in enumerate(conjuntos):
15        print(f"DNI {i+1}: {conjunto}")
16
17    if len(conjuntos) < 3:
18        print("Se requieren al menos 3 DNIs para todas las operaciones.")
19        return
20
21    A, B, C = conjuntos[:3] # Asignamos los primeros 3 conjuntos a A, B y C
22
23    print("\nOperaciones entre conjuntos:")
24    print("Unión (A ∪ B ∪ C):", A | B | C)
25    print("Intersección (A ∩ B ∩ C):", A & B & C)
26    print("Diferencia (A - B):", A - B)
27    print("Diferencia simétrica (A Δ B):", A ^ B)
28
29    print("\nFrecuencia de dígitos en cada DNI:") # define la frecuencia de dígitos en cada DNI
30    for i, dni in enumerate(dnis):
31        print(f"DNI {i+1}:")
32        for digito in "0123456789":
```

Activar Windows
Ve a Configuración para activar Windows.

```
33        print(f" Dígito {digito}: {dni.count(digito)} veces")
34
35    print("\nSuma de dígitos por DNI:") # calcula la suma de los dígitos de cada DNI
36    for i, dni in enumerate(dnis):
37        suma = sum(int(d) for d in dni)
38        print(f"DNI {i+1}: {suma}")
39
40    print("\nEvaluación de condiciones:") # evalúa condiciones sobre los conjuntos de dígitos
41    union_total = A | B | C
42    for dig in union_total:
43        if dig in A and dig in B and dig in C:
44            print(f"Dígito {dig}: compartido")
45
46    for i, conjunto in enumerate(conjuntos):
47        if len(conjunto) > 6:
48            print(f"DNI {i+1}: Diversidad numérica alta")
49
50 def operaciones_con_años(): # realiza operaciones con años de nacimiento
51    print("\n--- OPERACIONES CON AÑOS DE NACIMIENTO ---")
52    años_nacimiento = input("Ingrese los años de nacimiento separados por coma: ").split(",")
53    años_nacimiento = [int(a.strip()) for a in años_nacimiento]
54
55    pares = sum(1 for año in años_nacimiento if año % 2 == 0)
56    impares = len(años_nacimiento) - pares
57
58    print("Años pares:", pares)
59    print("Años impares:", impares)
60
61    if all(año > 2000 for año in años_nacimiento):
62        print("Grupo Z")
63
```

Activar Windows
Ve a Configuración para activar Windows.


```
64     if any(es_bisiesto(año) for año in años_nacimiento):
65         print("Tenemos un año especial")
66
67     año_actual = datetime.datetime.now().year
68     edades = [año_actual - año for año in años_nacimiento]
69
70     print("\nProducto cartesiano entre años y edades:")
71     for año in años_nacimiento:
72         for edad in edades:
73             print((año, edad))
74
75 def menu(): # muestra el menú principal y permite seleccionar operaciones
76     while True:
77         print("\n===== MENÚ PRINCIPAL =====")
78         print("1. Operaciones con DNIs")
79         print("2. Operaciones con años de nacimiento")
80         print("0. Salir")
81
82         opcion = input("Seleccione una opción: ")
83
84         if opcion == "1":
85             operaciones_con_dnis()
86         elif opcion == "2":
87             operaciones_con_años()
88         elif opcion == "0":
89             print("Programa finalizado.")
90             break
91         else:
92             print("Opción inválida. Intente de nuevo.")
93
```

Activar Windows
Ve a Configuración para activar Windows.