



Trabajo Integrador Final

Programación I

TEMA: Algoritmos de búsqueda y ordenamiento

Alumnos -

Laura Diaco (Laura.diaco@tupad.utn.edu.ar)

Matias Mansilla (matias.mansilla@tupad.utn.edu.ar)

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Organización Empresarial

Docente Titular

Julieta Trape

Docente Tutor

Tomas Ferro

Junio de 2025

Tabla de contenido

1. Introducción	3
2. Marco Teórico	5
3. Caso Práctico	10
4. Metodología Utilizada	19
5. Resultados Obtenidos	20
6. Conclusiones	23
7. Bibliografía	26

Introducción

En el mundo de la informática, la eficiencia para recuperar y organizar datos es crucial para el rendimiento de los sistemas. La creciente cantidad de información requiere métodos precisos que faciliten tanto la búsqueda de datos relevantes como su ordenación sistemática. Es aquí donde convergen dos pilares fundamentales de la programación: los algoritmos de búsqueda y los algoritmos de ordenamiento.

Los **algoritmos de búsqueda** tienen como objetivo localizar de manera rápida y precisa la información deseada en listas, bases de datos y otras estructuras. Dos enfoques ampliamente utilizados son la búsqueda lineal y la búsqueda binaria. La búsqueda lineal examina cada elemento de forma secuencial—tal como revisarías rincón por rincón de tu casa para encontrar las llaves—mientras que la búsqueda binaria divide sistemáticamente el conjunto de datos en mitades, permitiéndote ir directamente a la sección correcta, igual que buscar una palabra en un diccionario.

Por otro lado, el **ordenamiento** organiza los datos de acuerdo con criterios específicos, como el orden numérico o alfabético, y resulta esencial para optimizar tanto las búsquedas como otros procesos de análisis. Entre los algoritmos de ordenamiento más relevantes destacan **Bubble Sort**, que compara e intercambia elementos adyacentes hasta lograr la secuencia deseada; **Selection Sort**, que encuentra el elemento mínimo y lo sitúa al principio, repitiendo el proceso sobre el resto de la lista; **Insertion Sort**, que construye la solución insertando cada nuevo valor en su posición correcta dentro de la sublista ya ordenada; **Quick**



Sort, que divide recursivamente la lista mediante un pivote y ordena ambas particiones; y

Merge Sort, que fragmenta la lista en sublistas menores, las ordena individualmente y luego fusiona los resultados.

El interrelacionamiento entre búsqueda y ordenamiento es esencial: un conjunto de datos ordenado no solo permite la aplicación de métodos de búsqueda más eficientes—como la búsqueda binaria—sino que también agiliza el análisis y la gestión de la información. En este trabajo compararemos ambos enfoques en términos de eficiencia, complejidad computacional y aplicabilidad, implementando cada algoritmo en Python para observar su comportamiento en listas de distintos tamaños y distribuciones.

Marco Teórico

1. Concepto de Algoritmos de Búsqueda

Los algoritmos de búsqueda son fundamentales en informática, ya que permiten localizar elementos dentro de conjuntos de datos. Dependiendo de la estructura y organización de los datos, se pueden aplicar diferentes estrategias para optimizar la búsqueda y mejorar el rendimiento.

1. A. Búsqueda Lineal

La búsqueda lineal es el método más simple, en el que se revisan los elementos uno por uno hasta encontrar el objetivo. Es útil cuando los datos no están ordenados o cuando el tamaño de la lista es pequeño. Sin embargo, en conjuntos de datos grandes, su complejidad $O(n)$ la hace poco eficiente, ya que el tiempo de ejecución aumenta proporcionalmente con la cantidad de elementos.

1. B. Búsqueda Binaria

La búsqueda binaria es un algoritmo más eficiente, basado en la división del conjunto de datos en mitades. Sólo se puede aplicar en listas ordenadas, ya que compara el elemento central y descarta la mitad que no contiene el objetivo. Su complejidad $O(\log n)$ permite encontrar un elemento mucho más rápido que la búsqueda lineal en conjuntos grandes.

1.1 Comparación de Complejidades Algoritmos de búsqueda: Velocidad, Uso de Memoria y Escalabilidad

1.1. A. Velocidad en Algoritmos de Búsqueda

- Búsqueda Lineal: $O(n)$, crecimiento proporcional; adecuado para listas pequeñas o datos no ordenados.



Cuando un algoritmo es $O(n)$, duplicar el número de elementos casi duplica el tiempo de búsqueda.

- Búsqueda Binaria: $O(\log n)$, crecimiento logarítmico; cada comparación elimina la mitad de los datos restantes, ideal para grandes conjuntos ordenados.

Con $O(\log n)$, al duplicar el tamaño de la lista el número de pasos apenas aumenta en uno o dos.

1.1.B. Uso de Memoria en Algoritmos de Búsqueda

- Búsqueda Lineal ($O(1)$): sólo almacena unas cuantas variables, consumo constante.
- Búsqueda Binaria ($O(1)$): igualmente constante, pero requiere que la lista esté ordenada previamente.

1.1.C. Escalabilidad y Crecimiento en Algoritmos de Búsqueda

- Búsqueda Lineal ($O(n)$): no escala bien—cada aumento en n implica un aumento lineal en el tiempo.
- Búsqueda Binaria ($O(\log n)$): escala de forma eficiente—el tiempo crece muy lentamente al aumentar n .

2. Concepto de Algoritmos de Ordenamiento

Los algoritmos de ordenamiento organizan datos según un criterio específico (numérico, alfabético, etc.). Una vez ordenada la secuencia, las búsquedas y los análisis posteriores son más rápidos y sencillos.



2.A. Bubble Sort

Bubble Sort compara e intercambia elementos adyacentes si están fuera de lugar, realizando pasadas sucesivas hasta que la lista está ordenada. Su complejidad temporal es $O(n^2)$ en el peor y promedio de los casos, por lo que resulta poco práctico para grandes volúmenes de datos.

2.B. Selection Sort

Selection Sort busca el elemento mínimo en la sección no ordenada y lo coloca al principio, repitiendo esto hasta completar la ordenación. Reduce intercambios respecto a Bubble Sort, pero sigue teniendo $O(n^2)$ en todos los casos.

2.C. Insertion Sort

Insertion Sort inserta cada elemento en su posición correcta dentro de la sublista ya ordenada. En el mejor caso (lista casi ordenada) es $O(n)$, pero en el peor y promedio presenta $O(n^2)$.

2.D. Quick Sort

Quick Sort elige un pivote, partitiona la lista en elementos menores y mayores, y ordena recursivamente cada partición. Su complejidad promedio es $O(n \log n)$ y en el peor caso (pivote mal elegido) $O(n^2)$. Requiere $O(\log n)$ de espacio adicional para la recursión.

2.E. Merge Sort

Merge Sort divide la lista en mitades, ordena cada mitad recursivamente y luego fusiona ambas secuencias. Garantiza $O(n \log n)$ en todos los casos y necesita $O(n)$ de espacio extra para la fusión. Es estable, manteniendo el orden relativo de elementos iguales.

2.1 Comparación de Complejidades Algoritmos de Ordenamiento: Velocidad, Uso de Memoria y Escalabilidad



2.1.A. Velocidad en Algoritmos de Ordenamiento

- Bubble Sort: $O(n^2)$ en peor y promedio, $O(n)$ en el mejor caso (lista ya ordenada). Al duplicar el número de elementos, el total de comparaciones e intercambios se cuadriplica, por lo que su tiempo de ejecución crece muy rápidamente con n .
- Selection Sort: $O(n^2)$ en todos los casos. Recorre la lista completa para buscar el siguiente elemento mínimo en cada iteración, sin beneficiarse de un posible orden previo.
- Insertion Sort: $O(n^2)$ en peor y promedio, $O(n)$ en el mejor caso (datos casi ordenados). Si la lista está casi ordenada, solo realiza comparaciones sucesivas y desplaza pocos elementos, reduciendo drásticamente el tiempo.
- Quick Sort: $O(n \log n)$ en promedio y $O(n^2)$ en el peor caso (pivot mal elegido, por ejemplo siempre el primer o último elemento). Con una buena estrategia de selección de pivot (aleatorio o mediana de tres), el caso cuadrático se vuelve muy poco frecuente.
- Merge Sort: $O(n \log n)$ en todos los casos. Su división constante de la lista y la fusión garantizan un crecimiento moderado del tiempo de ejecución, independiente del orden inicial de los datos.

2.1.B. Uso de Memoria en Algoritmos de Ordenamiento

- Bubble Sort, Selection Sort e Insertion Sort: $O(1)$. Son algoritmos in-place que solo necesitan unas cuantas variables auxiliares para intercambios o desplazamientos.
- Quick Sort: $O(\log n)$. Además del espacio constante para swaps, utiliza pila de llamadas recursivas cuya profundidad promedio es $\log n$ (pero puede crecer a n en el peor caso).
- Merge Sort: $O(n)$. Requiere un arreglo auxiliar de tamaño n para guardar temporalmente los elementos durante el proceso de fusión, lo que implica costes de memoria lineales.

2.1.C. Escalabilidad y Crecimiento del Algoritmo

- Bubble Sort, Selection Sort e Insertion Sort ($O(n^2)$): no escalan bien. A medida que n crece, el tiempo de ordenación aumenta de forma cuadrática, volviéndolos inviables en listas grandes.
- Quick Sort ($O(n \log n)$ promedio): escala eficientemente en la mayoría de aplicaciones reales, ofreciendo un excelente equilibrio entre velocidad y uso moderado de memoria.
- Merge Sort ($O(n \log n)$): escalabilidad estable y predecible, ideal para volúmenes de datos muy grandes si se dispone del espacio extra para la fusión.

Caso Práctico

Algoritmos de búsqueda

Desarrollamos un código en Python que nos permita evaluar de modo concreto la forma en que ambos tipos de búsqueda se comportan dependiendo la cantidad de elementos que deben recorrer. En primera instancia realizamos la definición de funciones que nos permitirán realizar la búsqueda dentro de una lista de forma lineal o binaria

```
# Funciones de búsqueda
def busqueda_lineal(lista, objetivo):
    for producto in lista:
        if producto[0] == objetivo:
            return producto
    return None

def busqueda_binaria(lista, objetivo):
    inicio, fin = 0, len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio][0] == objetivo:
            return lista[medio]
        elif lista[medio][0] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return None
```

Como se puede apreciar en la función `busqueda_lineal` se recorre elemento a elemento hasta llegar al objetivo, mientras que en la función `busqueda_binaria` se delimita un medio que nos servirá como punto de referencia y desde el cual se verificará un extremo y otro de la lista con la variable `inicio= medio+1` o `inicio =medio-1`

```
# Lista de productos base
productos_base = ["Auriculares", "Celular", "Laptop", "Monitor", "Mouse", "Teclado", "Tablet"]

# Diferentes tamaños de inventario
tamanios = [1000, 5000, 10000, 50000, 100000]

for tam in tamanios:
    inventario = [] # Lista vacía

    # Generar productos con nombres y códigos
    for i in range(1, tam + 1):
        producto = productos_base[i % len(productos_base)] # Alternamos productos en orden
        inventario.append(f"{producto} {i}", 1000 + i)

    inventario.sort() # Ordenamos la lista alfabéticamente para búsqueda binaria
```

A continuación, realizamos la creación del inventario sobre el cual realizaremos las búsquedas. Debido a que necesitamos tamaños grandes para poder evaluar correctamente las dos formas de búsquedas sobre las que estamos trabajando asignamos a la variable “tamanios” los siguientes valores: 1000 , 5000, 10000,50000,100000. Por otro lado, a cada producto le asignamos un código y para realizarlo lo hacemos a través de producto = productos_base[i % len(productos_base)] que nos permite que los productos se asignen equitativamente ya que len(productos_base) devuelve la cantidad de elementos en la lista de productos (7 en este caso: "Auriculares", "Celular", "Laptop", "Monitor", "Mouse", "Teclado", "Tablet").

% len(productos_base) utiliza el operador módulo (%), que devuelve el resto de la división de i entre len(productos_base).

Esto permite que cuando i crece, el índice se repita en un ciclo continuo sobre la lista de productos. Logrando así que cada vez que i llega al final de la lista de productos, vuelve al inicio y se repite el patrón.

Luego con la función. append() generamos estos elementos en el inventario y con la función. sort() los ordenamos de forma alfabética.

```

# Producto específico a buscar
producto_a_buscar = inventario[len(inventario) // 2][0] # Tomamos un producto que sabemos que existe para que nos
# Medir tiempo de búsqueda lineal
inicio = time.time()
resultado_lineal = busqueda_lineal(inventario, producto_a_buscar)
fin = time.time()
print(f"Búsqueda Lineal en {tam} productos tomó {fin - inicio:.6f} segundos. Resultado: {resultado_lineal}")

# Medir tiempo de búsqueda binaria
inicio = time.time()
resultado_binaria = busqueda_binaria(inventario, producto_a_buscar)
fin = time.time()
print(f"Búsqueda Binaria en {tam} productos tomó {fin - inicio:.6f} segundos. Resultado: {resultado_binaria}")

```

Luego definimos el producto a buscar y en este caso lo realizamos con la siguiente porción de código: `producto_a_buscar = inventario[len(inventario) // 2][0]` ya que realizamos pruebas y debido a que el tamaño de los inventarios se va modificando nos quedaba fuera de rango y la búsqueda se realizaba y el resultado terminaba en `None`, por lo tanto decidimos modificarlo al código antes mencionado para asegurarnos que el producto realmente existe y esto es gracias a la posición del elemento que en este caso será “Monitor”

Por ultimo realizamos las mediciones de tiempo de ambos tipos de búsqueda para ello hemos importado al principio del código la librería `time` de Python que nos permite medir el tiempo de ejecución de las búsquedas y poder imprimir por pantalla el resultado de la búsqueda y el tiempo que le llevo encontrarlo, a continuación, mostraremos el resultado de su ejecución.

```

Búsqueda Lineal en 1000 productos tomó 0.000000 segundos. Resultado: ('Monitor 549', 1549)
Búsqueda Binaria en 1000 productos tomó 0.000000 segundos. Resultado: ('Monitor 549', 1549)
Búsqueda Lineal en 5000 productos tomó 0.000000 segundos. Resultado: ('Monitor 3237', 4237)
dos. Resultado: ('Monitor 5491', 6491)
Búsqueda Binaria en 10000 productos tomó 0.000000 segundos. Resultado: ('Monitor 5491', 6491)
Búsqueda Lineal en 5000 productos tomó 0.001000 segundos. Resultado: ('Monitor 325', 1325)
Búsqueda Binaria en 5000 productos tomó 0.000000 segundos. Resultado: ('Monitor 325', 1325)
Búsqueda Lineal en 100000 productos tomó 0.003999 segundos. Resultado: ('Monitor 54995', 55995)
Búsqueda Binaria en 100000 productos tomó 0.000000 segundos. Resultado: ('Monitor 54995', 55995)

```

En el mismo se ve como la búsqueda lineal funciona bien en listas pequeñas, pero se vuelve lenta en listas grandes. Mientras que la búsqueda binaria resultó mucho más rápida en listas

grandes porque no necesita revisar todos los elementos, sino que descarta la mitad en cada paso

Algoritmos de Ordenamiento

En esta sección, implementamos tres algoritmos fundamentales de ordenamiento en Python: Bubble Sort, Selection Sort e Insertion Sort. Cada uno sigue una estrategia distinta para organizar los datos, y analizamos su comportamiento al aplicarlos sobre listas de distintos tamaños.

```
1  import random
2  import time
3
4  # Funciones de ordenamiento
5
6  def bubble_sort(lista):
7      n = len(lista)
8      for i in range(n):
9          swapped = False
10         for j in range(0, n - i - 1):
11             if lista[j] > lista[j + 1]:
12                 lista[j], lista[j + 1] = lista[j + 1], lista[j]
13                 swapped = True
14         if not swapped:
15             break
16     return lista
17
18 def selection_sort(lista):
19     n = len(lista)
20     for i in range(n):
21         min_idx = i
22         for j in range(i + 1, n):
23             if lista[j] < lista[min_idx]:
24                 min_idx = j
25         lista[i], lista[min_idx] = lista[min_idx], lista[i]
26     return lista
27
28 def insertion_sort(lista):
29     for i in range(1, len(lista)):
30         key = lista[i]
31         j = i - 1
32         while j >= 0 and lista[j] > key:
33             lista[j + 1] = lista[j]
34             j -= 1
35         lista[j + 1] = key
36     return lista
```

```

38 def quick_sort(lista):
39     if len(lista) <= 1:
40         return lista
41     pivot = lista[len(lista) // 2]
42     menores = [x for x in lista if x < pivot]
43     iguales = [x for x in lista if x == pivot]
44     mayores = [x for x in lista if x > pivot]
45     return quick_sort(menores) + iguales + quick_sort(mayores)
46
47 def merge_sort(lista):
48     if len(lista) <= 1:
49         return lista
50     mid = len(lista) // 2
51     left = merge_sort(lista[:mid])
52     right = merge_sort(lista[mid:])
53     merged = []
54     i = j = 0
55     while i < len(left) and j < len(right):
56         if left[i] < right[j]:
57             merged.append(left[i]); i += 1
58         else:
59             merged.append(right[j]); j += 1
60     merged.extend(left[i:])
61     merged.extend(right[j:])
62     return merged

```

Para comenzar, importamos las librerías random y time, que nos permitirán generar listas aleatorias y medir el tiempo de ejecución de cada algoritmo. Luego, definimos las funciones de ordenamiento:

1. Bubble Sort: En esta función, se recorre la lista múltiples veces utilizando dos bucles.

El bucle externo controla el número total de pasadas, mientras que el bucle interno compara pares adyacentes e intercambia sus posiciones si están en el orden incorrecto. Además, utilizamos una variable swapped para detectar si se realizaron intercambios en



una pasada. Si en una iteración no hubo cambios, significa que la lista ya está ordenada y el algoritmo termina antes de completar todas las pasadas posibles.

2. **Selection Sort:** Aquí, el algoritmo utiliza dos bucles anidados. El bucle externo recorre la lista elemento por elemento, mientras que el bucle interno busca el elemento más pequeño dentro de la sección no ordenada de la lista. Una vez encontrado, se intercambia con la posición actual del bucle externo. De esta manera, el algoritmo va construyendo progresivamente la parte ordenada de la lista desde el inicio hasta el final.
3. **Insertion Sort:** En esta función, se toma cada elemento de la lista y se inserta en la posición correcta dentro de la parte ya ordenada de la lista. Para ello, se compara el elemento con los valores anteriores y, si es menor, se desplazan los elementos mayores hacia la derecha para hacerle espacio. Se utiliza un bucle while para mover los valores y reubicarlos correctamente hasta que el elemento encuentra su posición definitiva.
4. **Quick Sort:** La función quick_sort(lista) aplica el enfoque "divide y vencerás". Primero, comprueba si la lista tiene uno o ningún elemento, en cuyo caso la retorna porque ya está ordenada. Si no, selecciona un pivote (en este ejemplo, el elemento central de la lista). Luego, crea tres sublistas:
 - menores:** elementos menores que el pivote,
 - iguales:** elementos iguales al pivote,
 - mayores:** elementos mayores que el pivote.Posteriormente, la función se llama recursivamente para ordenar las sublistas de menores y mayores, y finalmente se concatena y retorna el resultado completo (menores ordenados + iguales + mayores ordenados). Este método suele tener una complejidad promedio de $O(n \log n)$, aunque en el peor caso puede degradarse a $O(n^2)$.
5. **Merge Sort:** La función merge_sort(lista) también utiliza la estrategia de "divide y vencerás". Comienza dividiendo la lista por la mitad y aplicándose recursivamente

sobre cada parte hasta que cada sublistas tenga un solo elemento (o ninguno), garantizando que estas ya están ordenadas. Posteriormente, la función fusiona las dos sublistas ordenadas mediante un proceso que compara sus elementos, combinándolos en una nueva lista final ordenada. El proceso de fusión recorre ambas sublistas y, al terminar, se agregan los elementos restantes. Merge Sort mantiene una complejidad de $O(n \log n)$ en todos los casos y es un algoritmo estable, aunque requiere espacio adicional para la fusión.

```

64     # Datos de prueba y medición de tiempos
65
66     tamanios = [1_000, 5_000, 10_000, 50_000, 100_000]
67     algoritmos = {
68         "Bubble Sort": bubble_sort,
69         "Selection Sort": selection_sort,
70         "Insertion Sort": insertion_sort,
71         "Quick Sort": quick_sort,
72         "Merge Sort": merge_sort
73     }
74
75     for n in tamanios:
76         # Generar lista aleatoria de n elementos y desordenarla
77         lista_base = list(range(n))
78         random.shuffle(lista_base)
79         print(f"\n== Tamaño: {n} elementos ==")
80
81         for nombre, funcion in algoritmos.items():
82             copia = lista_base.copy()
83             t1 = time.time()
84             resultado = funcion(copia)
85             t2 = time.time()
86             # Verificamos que quedó ordenada
87             assert resultado == sorted(lista_base)
88             print(f"{nombre:13s}: {t2 - t1:.4f} s")

```

A continuación, se define una lista de tamaños de prueba, “tamanios”, que contiene distintos valores (1,000; 5,000; 10,000; 50,000 y 100,000) para evaluar el rendimiento de varios

algoritmos de ordenamiento ante volúmenes crecientes de datos. Además, se crea un diccionario llamado algoritmos que asocia, por nombre, cada función de ordenamiento implementada (Bubble Sort, Selection Sort, Insertion Sort, Quick Sort y Merge Sort).

Para cada tamaño en la lista “tamanios”, el código genera una lista de números consecutivos usando `list(range(n))` y luego la desordena con `random.shuffle()`, simulando así un conjunto de datos en estado aleatorio. Se imprime el tamaño de la lista para contextualizar los resultados.

A continuación, se itera sobre cada algoritmo del diccionario. Para cada uno, se crea una copia de la lista desordenada para no modificar la original, se registra el tiempo justo antes y después de ejecutar la función de ordenamiento (usando `time.time()`) y se calcula la diferencia para obtener la duración de la ejecución. Además, se verifica con un assert que el resultado obtenido sea correcto comparándolo con la lista ordenada mediante la función `sorted()`.

Finalmente, se imprime el nombre del algoritmo junto con el tiempo de ejecución medido. Esta estructura permite comparar de manera práctica cómo varía el tiempo de ordenamiento en función del algoritmo empleado y del tamaño de los datos.

Por último, realizamos las mediciones de tiempo de los diferentes tipos de ordenamiento y para ello hemos importado al principio del código la librería `time` de Python que nos permite medir el tiempo de ejecución de las búsquedas y poder imprimir por pantalla el resultado del tiempo que le llevó ordenar el listado con distintas cantidades de elementos, a continuación, mostraremos el resultado de su ejecución:

```
== Tamaño: 1000 elementos ==
Bubble Sort : 0.1064 s
Selection Sort: 0.0477 s
Insertion Sort: 0.0435 s
Quick Sort : 0.0040 s
Merge Sort : 0.0054 s

== Tamaño: 5000 elementos ==
Bubble Sort : 2.7951 s
Selection Sort: 1.1317 s
Insertion Sort: 1.1144 s
Quick Sort : 0.0281 s
Merge Sort : 0.0323 s

== Tamaño: 10000 elementos ==
Bubble Sort : 10.8278 s
Selection Sort: 4.5300 s
Insertion Sort: 4.3893 s
Quick Sort : 0.0627 s
Merge Sort : 0.0670 s

== Tamaño: 50000 elementos ==
Bubble Sort : 305.0620 s
Selection Sort: 141.6943 s
Insertion Sort: 109.0592 s
Quick Sort : 0.2886 s
Merge Sort : 0.3573 s

== Tamaño: 100000 elementos ==
Bubble Sort : 1273.0703 s
Selection Sort: 704.9356 s
Insertion Sort: 439.0997 s
Quick Sort : 0.6739 s
Merge Sort : 0.7352 s
```

Los resultados muestran que los algoritmos tradicionales de ordenamiento (Bubble Sort, Selection Sort e Insertion Sort) incrementan sus tiempos de ejecución de forma drástica al aumentarse el tamaño del arreglo, lo que los hace inadecuados para manejo de grandes volúmenes de datos. En contraste, algoritmos más eficientes como Quick Sort y Merge Sort se mantienen notablemente rápidos incluso para conjuntos de datos muy extensos, destacándose como la opción idónea en entornos de alto rendimiento y aplicaciones en tiempo real.



Metodología Utilizada

Para este caso práctico, utilizamos la metodología experimental, ya que realizamos pruebas comparativas entre dos algoritmos de búsqueda para medir su eficiencia en diferentes tamaños de inventario.

Características de la Metodología Aplicada:

Experimentación → Ejecutamos los métodos de búsqueda lineal y binaria en listas de distintos tamaños para analizar su rendimiento. Y realizamos lo mismo con los algoritmos de ordenamiento Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort

Análisis de tiempos de ejecución → Medimos cuánto tarda cada algoritmo de búsqueda en encontrar el producto y evaluamos los resultados. Así como también el tiempo que tarda cada una de las 5 metodologías en ordenar el listado para evaluar su efectividad

Comparación de escalabilidad → Observamos cómo cambia la velocidad cuando aumentamos el número de productos en el inventario.

Validación → Nos aseguramos de que el producto a encontrar existe en todas las listas, evitando errores en la búsqueda. Y en el caso del ordenamiento, nos aseguramos de que realmente haya realizado el orden correspondiente en cada tamaño de lista

Esta metodología nos permite tomar decisiones basadas en datos reales, demostrando cuál algoritmo es más eficiente según el tamaño del inventario.

Resultados Obtenidos

1. Comparación de Tiempos de Ejecución en algoritmos de búsqueda

Se realizaron pruebas con inventarios de **diferentes tamaños** para comparar la eficiencia de **búsqueda lineal y binaria**.

Tabla de tiempos de ejecución:

Tamaño del Inventario	Búsqueda Lineal (seg)	Búsqueda Binaria (seg)
1,000 productos	0.000000	0.000000
5,000 productos	0.000000	0.000000
10,000 productos	0.000000	0.000000
50,000 productos	0.001000	0.000000
100,000 productos	0.003999	0.000000

2. Análisis de Resultados en algoritmos de búsqueda

- ◆ En **inventarios pequeños (1,000 productos)**, ambas búsquedas funcionan rápido, sin diferencias notables en tiempos de ejecución.
- ◆ A medida que el tamaño del inventario aumenta, la **búsqueda lineal tarda más**, mientras que la **búsqueda binaria mantiene tiempos casi nulos**.
- ◆ En **100,000 productos**, la diferencia es significativa: la búsqueda binaria sigue siendo rápida, mientras que la búsqueda lineal empieza a mostrar un mayor tiempo de ejecución.

En conclusión, los resultados obtenidos resaltan que, aunque en inventarios de tamaño reducido ambas técnicas son igual de eficientes, la búsqueda binaria demuestra ser superior cuando se manejan grandes volúmenes de datos. Este hallazgo es fundamental para aplicaciones en entornos competitivos, donde una respuesta rápida y fiable es esencial para optimizar la gestión de la información.

1. Comparación de Tiempos de Ejecución en algoritmos de ordenamiento

Se realizaron pruebas con arreglos de diferentes tamaños para evaluar la eficiencia de distintos algoritmos de ordenamiento. La siguiente tabla muestra los tiempos de ejecución medidos (en segundos):

Tamaño del Arreglo	Bubble Sort (seg)	Selection Sort (seg)	Insertion Sort (seg)	Quick Sort (seg)	Merge Sort (seg)
1,000	0.1064	0.0477	0.0435	0.0040	0.0054
5,000	2.7951	1.1317	1.1144	0.0281	0.0323
10,000	10.8278	4.5380	4.3893	0.0627	0.0670
50,000	305.0620	141.6943	109.0592	0.2886	0.3573
100,000	1273.0703	704.9356	439.0997	0.6739	0.7352

2. Análisis de Resultados de algoritmos de ordenamiento

- ◆ **Arreglos pequeños:** Para 1,000 elementos, todos los algoritmos muestran tiempos de ejecución reducidos, sin diferencias significativas que afecten su desempeño de forma notoria.
- ◆ **Crecimiento de los tiempos:** Al aumentar el tamaño del arreglo, se evidencia una marcada escalada en los tiempos de ejecución de los algoritmos básicos (Bubble, Selection e Insertion).



Sort), lo que pone de manifiesto su ineficiencia con grandes volúmenes de datos.

◆ **Eficiencia en grandes volúmenes:** En arreglos de 100,000 elementos, Quick Sort y Merge Sort se destacan por mantener tiempos de ejecución muy bajos, demostrando ser las técnicas preferidas para casos donde la rapidez y el rendimiento son cruciales.

En resumen, los resultados obtenidos indican que, aunque los algoritmos básicos puedan funcionar adecuadamente en pequeños conjuntos de datos, para aplicaciones que requieran procesar grandes volúmenes de información es fundamental optar por algoritmos eficientes como Quick Sort y Merge Sort. De esta manera, se asegura un manejo ágil y organizado de la información, optimizando el rendimiento del sistema y permitiendo una mayor competitividad en entornos exigentes.



Conclusión

Este trabajo resalta la importancia de combinar dos herramientas clave en la gestión de información: las técnicas de búsqueda y los algoritmos de ordenamiento. Al abordar estos aspectos, se pone de manifiesto cómo, de la mano, pueden transformar la forma en que interactuamos con grandes volúmenes de datos, haciendo que todo el proceso sea más ágil y efectivo.

Por un lado, las metodologías de búsqueda se han mostrado esenciales para acceder rápidamente a datos relevantes. Imaginemos la tarea de encontrar una aguja en un pajar: contar con una técnica de búsqueda poderosa es como disponer de un imán que te acerca lo que realmente necesitas, facilitando diagnósticos y decisiones basadas en información precisa y oportuna. Este enfoque no solo ahorra tiempo, sino que también potencia la fiabilidad de cualquier análisis.

Por otro lado, los algoritmos de ordenamiento juegan un rol igualmente crucial. No se trata simplemente de poner los datos en orden, sino de organizarlos de manera que se puedan manejar de forma eficiente y adaptable a distintas circunstancias. Herramientas como *quicksort* y *mergesort* destacan por su capacidad para procesar grandes conjuntos de información rápidamente, mientras que técnicas como el *insertion sort* pueden ser ideales en contextos con volúmenes reducidos o cuando se prioriza la simplicidad en la implementación. Esta diversidad de métodos permite elegir la técnica que mejor se adapte a cada situación, considerando factores como la cantidad y la naturaleza de los datos.



La verdadera fortaleza radica en la integración de ambos enfoques. La capacidad de localizar la información con precisión, combinada con la habilidad de ordenarla eficazmente, crea un círculo virtuoso que potencia tanto los procesos operativos como la toma de decisiones estratégicas. Por ejemplo, en áreas como el análisis financiero o la gestión de inventarios, disponer de un sistema que primero busque la información relevante y luego la organice de manera coherente se traduce en decisiones más rápidas y acertadas. Esta sinergia no solo mejora el rendimiento global del sistema, sino que también sienta las bases para futuras innovaciones y optimizaciones continuas.

A este análisis se le pueden sumar elementos adicionales que ponen de relieve la aplicabilidad práctica del enfoque. Por ejemplo, en industrias tan diversas como la logística, el comercio electrónico y el análisis financiero, se ha comprobado que la integración de técnicas de búsqueda y algoritmos de ordenamiento no solo optimiza procesos internos, sino que también mejora la experiencia del usuario y la capacidad de respuesta ante cambios en tiempo real. Para ilustrar este punto, imaginemos un sistema de gestión de inventarios en una gran cadena de distribución: una búsqueda eficiente permite localizar rápidamente la información sobre productos específicos, mientras que un algoritmo de ordenamiento idóneo organiza estos datos priorizando aquellos productos más demandados o con stock limitado. Este ejemplo muestra cómo el manejo ágil y organizado de la información marca una diferencia decisiva en entornos altamente competitivos.

Asimismo, es esencial mirar hacia el futuro y reconocer el impacto de las tendencias emergentes. La evolución hacia algoritmos híbridos, que combinan métodos tradicionales con avances en inteligencia artificial, está abriendo nuevas posibilidades para personalizar y optimizar la gestión de datos en tiempo real. Este avance permite desarrollar soluciones que se adaptan dinámicamente a las demandas del entorno, lo cual es especialmente valioso en sectores donde el análisis predictivo y la adaptabilidad son críticos para el éxito.

En conclusión, la sinergia entre técnicas avanzadas de búsqueda y algoritmos de ordenamiento no solo fortalece la eficiencia operativa, sino que también impulsa el desarrollo de sistemas robustos y escalables que responden a las exigencias de la era digital. Al integrar la precisión en la localización de la información con la capacidad de organizarla eficazmente, se generan soluciones integrales que facilitan análisis profundos, fomentan la innovación y aseguran una competitividad sostenida en un mercado en constante cambio.

Bibliografia

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009).** *Introduction to algorithms* (3rd ed.). MIT Press.
- Knuth, D. E. (1998).** *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley Professional.
- Sedgewick, R., & Wayne, K. (2011).** *Algorithms* (4th ed.). Addison-Wesley Professional.
- Bentley, J. L., & McIlroy, M. D. (1993).** Engineering a sort function. *Software: Practice and Experience*, 23(11), 1249–1265.
- Williams, J. W. J. (1964).** Algorithm 232: Heapsort. *Communications of the ACM*, 7(6), 347–348.
- Gill, S. K., Singh, V. P., Sharma, P., & Kumar, D. (2018).** A comparative study of various sorting algorithms. *International Journal of Advanced Studies of Scientific Research*, 4(1).