UNIVERSIDAD TECNOLÓGICA NACIONAL

TRABAJO PRÁCTICO INTEGRADOR

Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos:

Cristian Serna – sernachristian700@gmail.com

Emanuel Facundo Ruidiaz – emafruidiaz@gmail.com

Materia: Programación 1

Profesor: Cinthia Rigoni

Fecha de Entrega: 9 de junio de 2025

Índice

Introducción

Los algoritmos de búsqueda y ordenamiento son herramientas clave en la gestión y procesamiento de datos en programación. En este trabajo práctico, se abordan sus conceptos fundamentales mediante videos explicativos y ejercicios, estudiando técnicas como la búsqueda lineal, binaria y los principales métodos de ordenamiento. Además, se desarrollan algoritmos en Python para resolver problemas reales, permitiendo búsquedas rápidas y ordenamientos eficientes.

La correcta implementación de estos algoritmos es esencial para optimizar el manejo de grandes volúmenes de información y mejorar la toma de decisiones en diferentes contextos computacionales.

Marco Teórico

Búsqueda lineal

La búsqueda lineal es un método simple que consiste en recorrer secuencialmente una lista de elementos para encontrar uno específico. Es eficiente para listas pequeñas o desordenadas, con una complejidad de tiempo promedio y peor caso de O(n), donde n es la cantidad de elementos.

Búsqueda binaria

Este algoritmo requiere que la lista esté ordenada. Consiste en dividir repetidamente la lista en mitades y comparar el elemento buscado con el elemento central, eliminando la mitad en la que no puede estar el elemento. Su eficiencia radica en una complejidad de O(log n), haciéndola mucho más rápida que la búsqueda lineal en listas ordenadas.

Búsqueda de interpolación

Similar a la búsqueda binaria, la búsqueda de interpolación asume que los datos están distribuidos uniformemente y estima la posición probable del elemento buscado mediante interpolación. Es efectiva en listas ordenadas con distribución uniforme, con una complejidad esperada de O(log log n), pero puede ser ineficiente si los datos no cumplen con esta condición.

Búsqueda de hash

Este método utiliza una función hash para mapear los datos a ubicaciones específicas en una estructura de datos llamada tabla hash. Permite búsquedas, inserciones y eliminaciones en promedio en O(1), siendo muy eficiente para operaciones con grandes volúmenes de datos, aunque su rendimiento puede verse afectado por colisiones.

Ordenamiento por burbuja (Bubble Sort)

Consiste en repetidos pasadas a través de la lista, comparando elementos adyacentes y permutándolos si están en el orden incorrecto. Es uno de los algoritmos más sencillos, pero también uno de los menos eficientes, con una complejidad de $O(n^2)$.

Ordenamiento por selección (Selection Sort)

Este método encuentra el elemento más pequeño (o más grande) en la lista y lo coloca en la posición correspondiente, repitiendo el proceso para el resto de la lista. Aunque simple, su eficiencia tampoco es buena en listas grandes, con una complejidad de $O(n^2)$.

Ordenamiento por inserción (Insertion Sort)

Construye la lista ordenada de izquierda a derecha, insertando cada elemento en su posición adecuada. Es eficiente para listas pequeñas o casi ordenadas, con una complejidad promedio de $O(n^2)$, pero en listas grandes puede ser lento.

Ordenamiento rápido (QuickSort)

Es uno de los algoritmos más eficientes en la práctica, basado en la técnica divide y vencerás. Selecciona un pivote, reorganiza los elementos en relación con este y recursivamente ordena las sublistas. Su complejidad promedio es $O(n \log n)$, aunque puede deteriorarse en $O(n^2)$ en ciertos casos.

Ordenamiento por mezcla (Merge Sort)

También utiliza la estrategia divide y vencerás, dividiendo la lista en partes iguales, ordenando cada parte de forma recursiva y luego fusionándolas. Tiene una

complejidad consistente de O(n log n), siendo muy eficiente y estable en cuanto a rendimiento en listas grandes.

Caso Práctico

Algoritmos de búsqueda

1. Búsqueda Lineal

Técnica:
Busca secuencialmente por cada elemento en la lista hasta encontrar el objetivo o llegar al final. Es útil en listas no ordenadas.

Lista =

5	3	6	8	9	0

Objetivo: 8

```
lista = [5, 3, 6, 8, 9, 0]
objetivo = 8

def busqueda_lineal(lista, objetivo):
    i = 0
    while i < len(lista):
        if lista[i] == objetivo:
            return i
        i += 1
    return -1

resultado = busqueda_lineal(lista, objetivo)
print(f"El elemento {objetivo} está en la posición: {resultado}")
```

Ventajas:

- Simple de implementar.
- Funciona en listas no ordenadas.

Desventajas:

- Ineficiente en listas grandes, con O(n) en peor caso.
- No se aprovecha si la lista está ordenada.

2. Búsqueda Binaria

Técnica:
Divide repetidamente la lista ordenada en mitades, comparando el elemento central con el objetivo. Reduce el rango de búsqueda a la mitad en cada iteración, logrando eficiencia O(log n).
Lista =

1	2	5	7	9

Objetivo: 7

```python
lista = [1, 3, 5, 7, 9]
objetivo = 7

def busqueda_binaria(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
    return -1

resultado = busqueda_binaria(lista, objetivo)
print(f"El elemento {objetivo} está en la posición: {resultado}")
```

Ventajas:
- Muy eficiente en listas ordenadas, O(log n).

Desventajas:
- Solo funciona en listas ordenadas.

3. Búsqueda de interpolación

Técnica:
Similar a binaria, pero estima la posición probable del elemento basándose en una interpolación, útil cuando los datos están distribuidos uniformemente.

Lista=

10	20	30	40	50

Objetivo: 30

```
lista = [10, 20, 30, 40, 50]
objetivo = 40

def busqueda_interpolacion(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
    while bajo <= alto and objetivo >= lista[bajo] and objetivo <=
lista[alto]:
        if lista[alto] == lista[bajo]:
            return bajo
        # Estima la posición
        pos = bajo + int((alto - bajo) * (objetivo - lista[bajo]) /
(lista[alto] - lista[bajo]))
        if lista[pos] == objetivo:
            return pos
        elif lista[pos] < objetivo:
            bajo = pos + 1
        else:
            alto = pos - 1
    return -1

resultado = busqueda_interpolacion(lista, objetivo)
print(f"El elemento {objetivo} está en la posición: {resultado}")
```

Ventajas:

- Muy eficiente en listas distribuidas uniformemente, O(log log n).

Desventajas:

- Solo funciona en listas ordenadas y con distribución uniforme.

4. Búsqueda por hash

Técnica:
Utiliza una función hash para mapear datos a posiciones en una estructura de datos 'hash table'. Permite búsquedas en promedio en O(1).

Lista =

‘Manzana’ : ‘Fruta’	‘Lechuga’ : ‘Verdura’	‘Banana’ : ‘Fruta’	‘Tomate’ : ‘Fruta y Verdura’

Objetivo: ‘Manzana’

```
# Creando un inventario usando un diccionario (hash table)
inventario = {}
# Insertamos frutas en el inventario
inventario['Manzana'] = 'Fruta'
inventario['Lechuga'] = 'Verdura'
inventario['Banana'] = 'Fruta'
inventario['Tomate'] = 'Fruta y Verdura'

# Búsqueda: ¿La fruta 'Manzana' está en el inventario?
nombre_fruta = 'Manzana'
resultado = inventario.get(nombre_fruta, 'No encontrado')
print(f"Resultado de búsqueda: {resultado}")

# Búsqueda de una fruta no existente
nombre_fruta2 = 'Naranja'
resultado2 = inventario.get(nombre_fruta2, 'No encontrado')
print(f"Resultado de búsqueda: {resultado2}")
```

Ventajas:
- Búsquedas, inserciones y eliminaciones en promedio en O(1).
- Muy eficiente con grandes volúmenes de datos.

Desventajas:
- Posibles colisiones que disminuyen la eficiencia.
- Es necesario manejar rehashing en casos de muchas colisiones.

Algoritmos de ordenamiento

1. Ordenamiento por Burbuja

Técnica:
Funciona comparando cada elemento de la lista con el siguiente elemento y luego intercambiando los elementos si están en el orden incorrecto.

Lista =

5	3	6	9	0

```python
def burbuja_ord(lista):
    n = len(lista)
    # Recorrer todos los elementos de la lista
    for i in range(n - 1):
        # Últimos i elementos ya están en su lugar
        for j in range(0, n - i - 1):
            # Recorrer la lista de 0 a n-i-1. Intercambiar si el
elemento encontrado es mayor que el siguiente elemento.
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista

# Ejemplo de uso:
lista = [5, 3, 6, 8, 9, 0]
print(f"Lista original: {lista}")
lista_ordenada = burbuja_ord(lista)
print(f"Lista ordenada mediante técnica Burbuja: {lista_ordenada}")
```

Ventajas:
- Fácil de entender.
- Ideal para listas pequeñas.

Desventajas:
- Muy poco eficiente y lento en listas grandes.
- Cuando la lista está en orden inverso.

2. Ordenamiento por Selección

Técnica:
Funciona encontrando el elemento más pequeño de la lista y luego intercambiándolo con el primer elemento. Este proceso se repite hasta que todos los elementos de la lista estén ordenados.
Lista =

8	3	6	5	9	0

```
def seleccion_ord(lista):
    n = len(lista)
    # Recorrer todos los elementos de la lista
    for i in range(n):
        # Encontrar el elemento mínimo en el resto de la lista no
ordenada
        min = i
        for j in range(i + 1, n):
            if lista[j] < lista[min]:
                min = j
        # Intercambiar el elemento mínimo encontrado con el primer
elemento no ordenado
        lista[i], lista[min] = lista[min], lista[i]
    return lista

# Ejemplo de uso:
lista = [8, 3, 6, 5, 9, 0]
print(f"Lista original: {lista}")
lista_ordenada = seleccion_ord(lista)
print(f"Lista ordenada mediante técnica Selección: {lista_ordenada}")
```

Ventajas:
- Eficiente en listas pequeñas.
- Fácil de entender e implementar

Desventajas:
- Poco eficiente y lento en listas grandes y desordenadas.
- El orden relativo de los elementos con valores iguales puede no conservarse.

3. Ordenamiento por Inserción

Técnica:
Funciona insertando cada elemento de la lista en su posición correcta en la lista ordenada.

Lista =

9	3	0	8	5	6

```
def insercion_ord(lista):
    # Recorrer desde el segundo elemento hasta el final
    for i in range(1, len(lista)):
        key = lista[i]
        j = i - 1
        # Mover los elementos de lista que son mayores que la key, a
una posición adelante de su posición actual
        while j >= 0 and key < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = key
    return lista

# Ejemplo de uso:
lista = [9, 3, 0, 8, 5, 6]
print(f"Lista original: {lista}")
lista_ordenada = insercion_ord(lista)
print(f"Lista ordenada mediante técnica Inserción: {lista_ordenada}")
```

Ventajas:

- Útil cuando la lista está casi ordenada.
- Eficiente en listas pequeñas.
- mantiene el orden relativo original de los elementos que tienen el mismo valor.

Desventajas:

- Poco útil cuando la lista está en orden inverso.
- Poco eficiente en listas grandes.

4. Ordenamiento rápido (QuickSort)

Técnica:
Funciona seleccionando un elemento pivot y particionando la lista en dos sublistas: una con elementos menores que el **pivot** y otra con elementos mayores. Luego, se aplica Quicksort recursivamente a cada sublista.

Lista =

5	3	6	8	9	0

```
def quicksort_ord(lista):
    if len(lista) <= 1:
        return lista

    pivot = lista[len(lista) // 2]  # Elegir un pivote (aquí el del
medio)
    izquierda = [x for x in lista if x < pivot]
    mitad = [x for x in lista if x == pivot]
    derecha = [x for x in lista if x > pivot]

    return quicksort_ord(izquierda) + mitad + quicksort_ord(derecha)

# Ejemplo de uso:
lista = [5, 3, 6, 8, 9, 0]
print(f"Lista original: {lista}")
lista_ordenada = quicksort_ord(lista)
print(f"Lista ordenada con Quicksort: {lista_ordenada}")
```

Ventajas:

- Eficiente y rapido en listas grandes.
- No requiere memoria adicional

Desventajas:

- Pierde efectividad cuando la lista está ordenada.
- Implementación un poco complicada

5. Ordenamiento por Mezcla (MergeSort)

Técnica:
Funciona dividiendo la lista en dos partes, ordenando cada parte y luego fusionando las dos partes ordenadas.

Lista =

8	6	3	5	9	0

```python
def merge_ord_mitades(lista):
    if len(lista) <= 1:
        return lista

    # Dividir la lista en dos mitades, izquierda y derecha
    mitad = len(lista) // 2
    mitad_izquierda = lista[:mitad]
    mitad_derecha = lista[mitad:]

    # Ordenar ambas mitades
    mitad_izquierda = merge_ord_mitades(mitad_izquierda)
    mitad_derecha = merge_ord_mitades(mitad_derecha)

    # Unir las mitades izquierda y derecha ordenadas
    return merge(mitad_izquierda, mitad_derecha)

#Función final de ordenamiento
def merge(izquierda, derecha):
    lista_ord_merge = []
    i = 0
    j = 0

    while i < len(izquierda) and j < len(derecha):
        if izquierda[i] < derecha[j]:
            lista_ord_merge.append(izquierda[i])
            i += 1
        else:
            lista_ord_merge.append(derecha[j])
            j += 1

    # Añadir los elementos restantes
    while i < len(izquierda):
        lista_ord_merge.append(izquierda[i])
        i += 1
```

```
    while j < len(derecha):
        lista_ord_merge.append(derecha[j])
        j += 1

    return lista_ord_merge

# Ejemplo de uso:
lista = [8, 6, 3, 5, 9, 0]
print(f"Lista original: {lista}")
lista_ordenada = merge_ord_mitades(lista)
print(f"Lista ordenada con Merge Sort: {lista_ordenada}")
```

Ventajas:

- Mantiene el orden relativo de los elementos iguales en la lista original.
- Efectivo para conjuntos de datos que se pueden acceder secuencialmente, como arreglos, vectores y listas ligadas.

Desventajas:

- Requiere un espacio adicional de memoria, esto puede ser un problema cuando se trabaja con una gran cantidad de datos y recursos limitados.
- Para listas muy pequeñas, la sobrecarga de dividir y combinar puede hacer que otros algoritmos más simples sean ligeramente más eficientes.

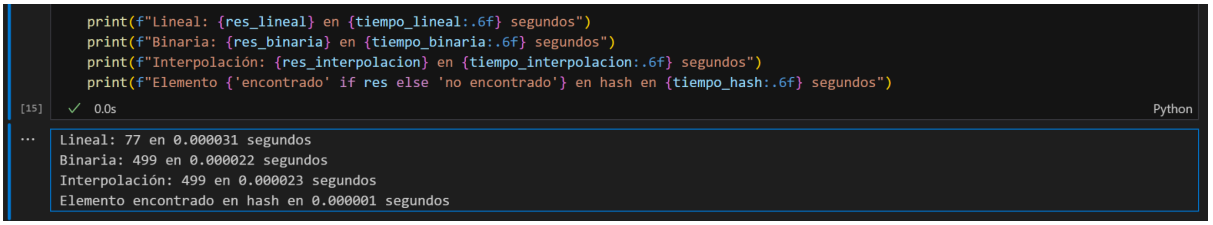
Metodología Utilizada

Para esta presentación utilizamos la siguiente metodología:

- Investigación previa
    - Algoritmos de búsqueda.
    - Algoritmos de ordenamiento.
    - Eficiencia de los mismos.
    - Ventajas y desventajas.
- Explicación de los algoritmos
    - Breve explicación del funcionamiento de los algoritmos de búsqueda y ordenamiento.
- Implementación del código
    - Se codificaron los algoritmos seleccionados utilizando el lenguaje de programación Python, con el objetivo de evaluar su funcionamiento real.
- Comparación de los algoritmos
    - Se aplicaron los algoritmos a ejemplos prácticos y se comparó su rendimiento, analizando su comportamiento frente a distintos tamaños y tipos de datos.
- Desarrollo de una conclusión
    - A partir de los resultados obtenidos, se elaboraron conclusiones sobre la eficiencia y adecuación de cada algoritmo en diferentes escenarios.

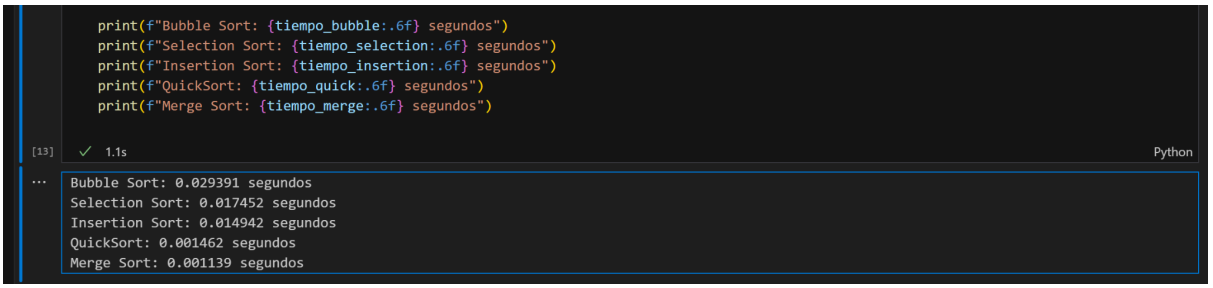
Resultados Obtenidos

Algoritmos de búsqueda:



Algoritmo	Tiempo promedio (segundos)	Descripción	Observaciones
Búsqueda lineal	0.000031	Busca secuencial en la lista	Lenta en listas grandes, necesita recorrer toda la lista
Búsqueda binaria	0.000022	Divide y vencerás en lista ordenada	Muy eficiente en listas ordenadas
Búsqueda de interpolación	0.000023	Aproxima la posición basada en el valor	Rápida en listas uniformemente distribuidas
Búsqueda de hash	0.000001	Usa tabla hash para búsquedas directas	La más rápida en búsquedas, requiere estructura adicional

Algoritmos de ordenamiento:



```
    print(f"Bubble Sort: {tiempo_bubble:.6f} segundos")
    print(f"Selection Sort: {tiempo_selection:.6f} segundos")
    print(f"Insertion Sort: {tiempo_insertion:.6f} segundos")
    print(f"QuickSort: {tiempo_quick:.6f} segundos")
    print(f"Merge Sort: {tiempo_merge:.6f} segundos")
```

[13] ✓ 1.1s Python

```
Bubble Sort: 0.029391 segundos
Selection Sort: 0.017452 segundos
Insertion Sort: 0.014942 segundos
QuickSort: 0.001462 segundos
Merge Sort: 0.001139 segundos
```

Algoritmo	Tiempo promedio (segundos)	Descripción	Observaciones
Ordenamiento por Burbuja	0.029391	Comparaciones repetidas, intercambios	Muy ineficiente en listas grandes, solo educativo
Ordenamiento por Selección	0.017452	Selecciona mínimo y lo coloca en orden	Similar a Bubble, lento en grandes conjuntos
Ordenamiento por Inserción	0.014942	Inserta elementos en posición correcta	Bueno para listas casi ordenadas
Ordenamiento rápido	0.001462	Divide y conquista, en promedio muy eficiente	Rápido, pero puede ser peor en casos específicos
Ordenamiento por mezcla	0.001139	Divide y combina eficientemente, estable	Bueno en archivos grandes, requiere espacio adicional

Conclusiones

En la selección de algoritmos, la eficiencia y la aplicabilidad dependen del contexto y las características de los datos.

Para búsquedas, el método de búsqueda de hash y búsqueda binaria ofrecen tiempos de respuesta significativamente bajos en promedio, siendo ideales para sistemas que requieren rapidez en consultas frecuentes y listas previamente ordenadas, respectivamente. La búsqueda lineal, aunque sencilla de implementar, solo es viable en listas pequeñas o datos no ordenados.

En cuanto al ordenamiento, métodos como Quicksort y Merge Sort destacan por su excelente rendimiento en listas grandes, alcanzando tiempos promedio muy bajos, que los hacen preferibles en la mayoría de las aplicaciones prácticas. Sin embargo, el método Merge sort requiere de un espacio adicional de memoria. En cuanto a los métodos Bubble, Selection, e Insertion son más simples pero considerablemente más lentos, por lo que se recomienda sólo en casos educativos o conjuntos de datos muy pequeños o casi ordenados.

En resumen, para proyectos que involucren grandes volúmenes de datos, es recomendable optar por algoritmos eficientes y optimizados como Quicksort, Merge Sort, y búsqueda binaria o hash, ya que ofrecen mejores resultados en términos de velocidad y rendimiento.

Bibliografía

Álvarez Bravo, Jose Vicente. “TEMA 6: ALGORITMOS DE BÚSQUEDA Y ORDENACIÓN.” *ALGORITMOS DE BÚSQUEDA Y ORDENACIÓN.* https://www.infor.uva.es/~jvalvarez/docencia/pII%20antiguo/tema6.pdf.

UTN - TUPAD. *Búsqueda y Ordenamiento en Programación*. 2025. *Búsqueda y Ordenamiento en Programación*, https://tup.sied.utn.edu.ar/mod/resource/view.php?id=3434.

Anexos

- Anexo A: Código fuente de algoritmos de búsqueda
- Anexo B: Código fuente de algoritmos de ordenamiento
- Anexo C: Datos utilizados en pruebas de rendimiento
- Anexo D: Tabla de tiempos de ejecución (en segundos)
- Video explicativo: Algoritmos de Búsqueda y Ordenamiento - Programación 1