Trabajo Integrador - Propuesta de Investigación - Programación I

Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos

Santiago Caiciia Masello - scaiciia@gmail.com
Macarena Cantoni - macacantoni@gmail.com

Profesor: Ariel Enferrel
Fecha de Entrega: 09/06/2025

Índice

Introducción

Los algoritmos de búsqueda y ordenamiento constituyen pilares fundamentales en el desarrollo de software, proporcionando soluciones eficientes, escalables y precisas para la gestión de información, contribuyendo así a la creación de programas más rápidos, organizados y confiables.

Se seleccionó el tema de los algoritmos de búsqueda y ordenamiento debido a su relevancia universal en la informática y el desarrollo de software. Estos algoritmos no solo son esenciales para optimizar el rendimiento y la eficiencia en el manejo de datos, sino que también constituyen una base fundamental para comprender estructuras de datos y principios algorítmicos.

Para finalizar, como caso práctico, se desarrolló un programa en Python que permite generar una lista de estudiantes con notas aleatorias, ordenar dicha lista utilizando métodos de ordenamiento como Burbuja y Quicksort, y realizar búsquedas (secuencial o binaria) de estudiantes por nombre. Esta implementación no solo demuestra la funcionalidad y eficiencia de los algoritmos en situaciones reales, sino que también permite comparar su rendimiento y utilidad en contextos concretos de manejo de datos.

Marco Teórico

La **búsqueda** es una operación fundamental en programación que se utiliza para encontrar un elemento específico dentro de un conjunto de datos.

Tipos de búsqueda

- **Búsqueda lineal:** Es el algoritmo de búsqueda más simple, recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Es fácil de implementar, pero puede ser lento para conjuntos de datos grandes.
- **Búsqueda binaria:** Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.
- **Búsqueda de interpolación:** Es un algoritmo de búsqueda que mejora la búsqueda binaria al estimar la posición del elemento deseado en función de su valor. Puede ser más eficiente que la búsqueda binaria para conjuntos de datos grandes con una distribución uniforme de valores.
- **Búsqueda de hash:** Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

Los dos métodos más comunes son:

Búsqueda lineal:

- Se recorre cada elemento de la lista hasta encontrar el deseado o llegar al final.
- Ventajas:
    - Sencillez: La búsqueda lineal es uno de los algoritmos de búsqueda más simples y fáciles de implementar. Solo requiere iterar a través de la lista de elementos uno por uno hasta encontrar el objetivo.
    - flexibilidad: La búsqueda lineal puede aplicarse a cualquier tipo de lista, independientemente de si está ordenada o no.
- Desventajas:
    - Ineficiencia en listas grandes: La principal desventaja de la búsqueda lineal es su ineficiencia en listas grandes. Debido a que compara cada elemento uno por uno, su tiempo de ejecución crece de manera lineal con el tamaño de la lista.
    - No es adecuada para listas ordenadas: Aunque puede funcionar en listas no ordenadas, la búsqueda lineal no es eficiente para listas ordenadas. En tales casos, algoritmos de búsqueda más eficientes, como la búsqueda binaria, son preferibles.

Búsqueda binaria:

- Funciona solo en listas ordenadas.
- Divide la lista en dos partes y busca en la mitad correspondiente, reduciendo el tamaño del problema con cada paso.
- Ventajas:
    - Eficiencia de listas ordenadas: La principal ventaja de la búsqueda binaria es su eficiencia en listas ordenadas. Su tiempo de ejecución es de O(log n), lo que significa que disminuye rápidamente a medida que el tamaño de la lista aumenta.
    - Menos comparaciones: Comparado con la búsqueda lineal, la búsqueda binaria realiza menos comparaciones en promedio, lo que lo hace más rápido para encontrar el objetivo.
- Desventajas:
    - Requiere una lista ordenada: La búsqueda binaria sólo es aplicable a listas ordenadas, Si la lista no está ordenada, se debe realizar una operación adicional para ordenarla antes de usar la búsqueda binaria.
    - Mayor complejidad de implementación: Comparado con la búsqueda lineal, la búsqueda binaria es más compleja de implementar debido a su naturaleza recursiva.

Aplicaciones

- Búsqueda de palabras clave en un documento
- Búsqueda de archivos en un sistema de archivos
- Búsqueda de registros en una base de datos
- Búsqueda de la ruta más corta en un gráfico
- Búsqueda de soluciones a problemas de optimización

Complejidad y medición de eficiencia:

La complejidad medida con O(n) es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada.

La notación O(n) se utiliza para describir el peor caso de complejidad de tiempo de un algoritmo. Esto significa que el algoritmo nunca tardará más de O(n) tiempo en ejecutarse para cualquier entrada de tamaño n.

La complejidad medida con O(n) es una herramienta útil para comparar diferentes algoritmos y elegir el más eficiente para una tarea determinada.

Algoritmo	Tiempo de ejecución	Detalle
Búsqueda lineal	O(n)	• El tiempo de búsqueda es directamente proporcional al tamaño de la lista. • Si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado.
Búsqueda binaria	O(log n)	• El tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. • Si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.

Los algoritmos de **ordenamiento**, son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla. Los datos son organizados de acuerdo a un criterio, como de menor a mayor o alfabéticamente.

Algoritmos de ordenamiento más comunes:

- **Ordenamiento por burbuja (Bubble Sort):** Simple y fácil de implementar. Funciona comparando cada elemento de la lista con el siguiente elemento y luego intercambiando los elementos si están en el orden incorrecto.
    - Ventajas:
        - Simplicidad: El algoritmo de burbuja es fácil de entender e implementar, lo que lo convierte en una buena opción para introducir conceptos de ordenamiento en la programación.
        - Implementación sencilla: Requiere poca cantidad de código y no involucra estructuras de datos complejas.
    - Desventajas:
        - Lento para listas grandes: Debido a su complejidad cuadrática el algoritmo de burbuja se vuelve lento en la práctica para listas de tamaño considerable.
        - No considera el orden parcial: A diferencia de otros algoritmos, el algoritmo de burbuja realiza el mismo número de comparaciones e intercambios sin importar si la lista ya está en gran parte ordenada.
- **Ordenamiento por selección (Selection Sort):** 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.

    - Ventajas:
        - Sencillez y facilidad de implementación.
        - Eficiencia para listas pequeñas o listas que ya estén parcialmente ordenadas.
    - Desventajas:
        - Baja eficiencia en listas de gran tamaño, ya que su complejidad temporal es cuadrática.
        - El algoritmo siempre realiza el mismo número de comparaciones e intercambios, incluso si la lista ya está ordenada.
- **Ordenamiento por inserción (Insertion Sort):** Funciona insertando cada elemento de la lista en su posición correcta en la lista ordenada.
    - Ventajas:
        - Baja sobrecarga: Requiere menos comparaciones y movimientos que algoritmos como el ordenamiento de burbuja, lo que lo hace más eficiente en términos de intercambios de elementos.
        - Simplicidad: el ordenamiento por inserción es uno de los algoritmos de ordenamiento más simples de implementar y entender. Esto lo hace adecuado para enseñar conceptos básicos de ordenamiento.
    - Desventajas:
        - Ineficiencia en listas grandes: A medida que el tamaño de la lista aumenta, el rendimiento del ordenamiento por inserción disminuye. Su complejidad cuadrática de O(n^2) en el pero caso lo hace ineficiente para las listas grandes.
        - No escalable: Al igual que otros algoritmos de complejidad cuadrática, el ordenamiento por inserción no es escalable para listas grandes, ya que su tiempo de ejecución aumenta considerablemente con el tamaño de la lista.
- **Ordenamiento rápido (Quicksort) :** Funciona dividiendo la lista en dos partes y luego ordenando cada parte de forma recursiva.
    - Ventajas:
        - Eficiencia promedio: En la mayoría de los casos, Quicksort es muy eficiente, con una complejidad de O(n log n), lo que lo hace ideal para ordenar grandes conjuntos de datos.
        - Uso eficiente de memoria: Al ser un algoritmo que se ejecuta in-place, Quicksort no requiere espacio adicional significativo, lo que lo hace más eficiente en términos de uso de memoria en comparación con otros algoritmos.
        - Velocidad en la práctica: En muchos escenarios, Quicksort tiende a ser más rápido que otros algoritmos de ordenación, especialmente cuando se implementa con una buena estrategia para elegir el pivote.
        - Facilidad de implementación: La implementación de Quicksort es relativamente simple y directa, lo que lo hace fácil de entender y de adaptar a diversos lenguajes de programación.
    - Desventajas:
        - Sensibilidad al pivote: La eficiencia de Quicksort depende en gran medida de la elección del pivote. Si el pivote se elige mal, especialmente en casos extremos como cuando el arreglo ya está ordenado o casi ordenado, la eficiencia puede degradarse a $O(n^2)$.

        - Inestabilidad: Quicksort no es estable en su implementación básica, lo que significa que no necesariamente conserva el orden relativo de los elementos con claves iguales.
        - Peor caso de complejidad: Aunque en promedio Quicksort tiene una complejidad de O(n log n), en el peor de los casos puede degenerar a $O(n^2)$, lo que lo hace menos deseable en situaciones donde se requiere garantizar tiempos de ejecución consistentemente buenos.
- **Ordenamiento por mezcla(Merge sort):** Funciona dividiendo la lista en dos partes, ordenando cada parte y luego fusionando las dos partes ordenadas.
    - Ventajas:
        - Eficiencia en todo tipo de casos: El algoritmo Merge Sort tiene una complejidad de tiempo de O(n log n) en todos los casos, lo que lo hace muy eficiente incluso para conjuntos de datos grandes y desordenados. Esta eficiencia lo hace especialmente útil en aplicaciones donde se necesita ordenar grandes cantidades de datos.
        - Estabilidad: Merge Sort es un algoritmo estable, lo que significa que conserva el orden relativo de los elementos con claves iguales. Esto lo hace útil en situaciones donde la estabilidad del ordenamiento es importante.
        - Implementación sencilla y comprensible: La lógica detrás del algoritmo Merge Sort es relativamente simple de entender, ya que se basa en el concepto de "dividir y conquistar". Esto facilita su implementación y depuración, lo que lo hace adecuado para enseñar y aprender sobre algoritmos de ordenamiento.
    - Desventajas:
        - Merge Sort requiere memoria auxiliar para almacenar las sublistas temporales durante el proceso de combinación. Esta necesidad de memoria adicional puede ser una desventaja en sistemas con restricciones de memoria.
        - No es óptimo para listas pequeñas: Aunque Merge Sort es muy eficiente para grandes conjuntos de datos, su desempeño puede ser peor que otros algoritmos de ordenamiento, como el algoritmo de inserción, para listas pequeñas o casi ordenadas. Esto se debe al costo adicional de dividir y combinar las sublistas.

¿En que basar la elección del algoritmo de ordenamiento adecuado?

- Tamaño de la lista.
- El tipo de datos.
- Requisitos de rendimiento.

¿Por qué es importante usar este tipo de algoritmos?

- **Eficiencia:** Mejoran el tiempo de ejecución de programas que manejan grandes cantidades de datos.
- **Organización:** Facilitan la estructuración y presentación de datos de manera coherente, simplificando su análisis y comprensión.
- **Escalabilidad:** Su diseño permite manejar conjuntos de datos de diferentes tamaños, adaptándose a las necesidades cambiantes de un programa.
- **Precisión:** Garantizan la recuperación de resultados exactos y relevantes, evitando errores y ambigüedades en la búsqueda de información.
- **Versatilidad:** Se aplican en una amplia gama de contextos y dominios, desde bases de datos y sistemas de archivos hasta aplicaciones web y motores de búsqueda.

Caso práctico

1. Se implementa las funciones de búsqueda (método secuencial y binaria)

```python
# Funcion que realiza una búsqueda secuencial en una lista de tuplas
def busqueda_secuencial(lista, objetivo):
    for alumno, nota in lista:
        if alumno == objetivo:
            return (alumno, nota)
    return None
```

```python
# Función para realizar una búsqueda binaria en una lista de tuplas orde
nadas por el primer elemento de cada tupla.
def busqueda_binaria(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio][0] == objetivo:
            return lista[medio]
        elif lista[medio][0] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
    return None
```

2. También las funciones de ordenamiento (método burbuja y quicksort)

```python
# Funcion de ordenamiento por metodo burbuja
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j][0] > lista[j+1][0]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
```

```python
# Funcion de ordenamiento por metodo Quicksort
def ordenamiento_quick(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[len(lista) // 2]
    izquierda = [x for x in lista if x[0] > pivote[0]]


"""Se modifica ya que la original no maneja los elementos iguales al p
ivote
    (es decir, si hay tuplas con el mismo valor en [0] que el pivote,
se perderán)."""
    iguales = [x for x in lista if x[0] == pivote[0]]

    derecha = [x for x in lista if x[0] < pivote[0]]
    return ordenamiento_quick(izquierda) + iguales +
ordenamiento_quick(derecha)
```

3. Se implementa el código que genera la lista de alumnos-nota.

```python
import os
import random
import time
from busquedaBinaria import busqueda_binaria
from busquedaSecuencial import busqueda_secuencial
from ordenamientoBurbuja import ordenamiento_burbuja
from ordenamientoQuicksort import ordenamiento_quick

"""
    Generacion de la lista de estudiantes con sus respectivas notas.
    Se generan n cantidad de estudiantes con nombres aleatorios y nota
s entre 0.0 y 10.0.
"""
def generar_estudiantes(n):
    nombres = [
        "Ana", "Luis", "Carlos", "María", "Jorge", "Sofía", "Pedro",
"Valentina", "Martín", "Camila",
        "Lucía", "Mateo", "Paula", "Diego", "Julieta", "Agustín",
"Micaela", "Tomás", "Florencia", "Santiago",
        "Emilia", "Benjamín", "Victoria", "Juan", "Josefina", "Lucas"
, "Milagros", "Facundo", "Rocío", "Nicolás",
        "Malena", "Gabriel", "Renata", "Franco", "Martina", "Simón",
"Guadalupe", "Ramiro", "Lara", "Axel",
        "Candelaria", "Bruno", "Ailén", "Iván", "Abril", "Federico",
"Luz", "Sebastián", "Bianca", "Gonzalo",
        "Ariana", "Matías", "Elena", "Leandro", "Agustina", "Delfina"
, "Maximiliano", "Carolina", "Emmanuel", "Sol",
        "Valeria", "Ignacio", "Celeste", "Esteban", "Juliana",
"Lautaro", "Pilar", "Thiago", "Aitana", "Ezequiel",
        "Mara", "Lisandro", "Ariadna", "Damián", "Aldana", "Joaquín",
"Catalina", "Alan", "Mara", "Enzo",
        "Noelia", "Ulises", "Tamara", "Kevin", "Melina", "Oscar",
"Aixa", "Ricardo", "Brenda", "Eduardo",
        "Selena", "Raúl", "Miriam", "Hugo", "Patricia", "Rubén",
"Lorena", "Marcos", "Daniela", "Fernando"
    ]
    estudiantes = [(random.choice(nombres), round(random.uniform(0.0,
10.0), 1)) for _ in range(n)]
    return estudiantes
```

4. Y funciones búsqueda y ordenar donde se encuentran las funciones implementadas anteriormente.

```
"""
    Función para buscar un estudiante en una lista.
    Dependiendo de si la lista está ordenada o no, se utiliza búsqueda
secuencial o binaria.
"""
def busqueda(lista, lista_ordenada):
    dato = input("Ingrese el nombre del estudiante a buscar: ")
    respuesta = None
    if (len(lista_ordenada) == 0):
        inicio = time.perf_counter()
        respuesta = busqueda_secuencial(lista, dato)
        fin = time.perf_counter()
        print(f"Demora: {fin - inicio:.10f} segundos")
    else:
        inicio = time.perf_counter()
        respuesta = busqueda_binaria(lista_ordenada, dato)
        fin = time.perf_counter()
        print(f"Demora: {fin - inicio:.10f} segundos")
    return respuesta

"""
    Función para ordenar una lista de estudiantes.
    Permite elegir entre dos métodos de ordenamiento: Burbuja (descend
ente) y Quicksort (ascendente).
"""
def ordenar(lista, lista_ordenada):
    while True:
        os.system('cls' if os.name == 'nt' else 'clear')
        print("Seleccione el método de ordenamiento:")
        print("1. Ascendente (Burbuja)")
        print("2. Descendente (Quicksort)")
        print("3. Atrás")
        opcion = input("Ingrese su opción: ")

        if opcion == "1":
            inicio = time.perf_counter()
            lista_ordenada = ordenamiento_burbuja(lista)
            fin = time.perf_counter()
            print(f"Demora: {fin - inicio:.10f} segundos")
            for nombre, nota in lista_ordenada:
                print(f"Nombre: {nombre}, Nota: {nota}")
            break
        elif opcion == "2":
            inicio = time.perf_counter()
            lista_ordenada = ordenamiento_quick(lista)
            fin = time.perf_counter()
            print(f"Demora: {fin - inicio:.10f} segundos")
            for nombre, nota in lista_ordenada:
                print(f"Nombre: {nombre}, Nota: {nota}")
            break
        elif opcion == "3":
            break
        else:
            print("Opción no válida.")
    return lista_ordenada
```

La búsqueda que se realice, depende de si la lista fue ordenada anteriormente o no. Cada uno de los diferentes métodos de “búsqueda” y “ordenar” miden el tiempo de demora de su ejecución.

5. Por último, se implementa el menú en el programa principal.

```python
# Programa principal
n = 10000  # Número de estudiantes a generar
estudiantes = generar_estudiantes(n)
lista_ordenada = []
while True:
    os.system('cls' if os.name == 'nt' else 'clear')
    print("-----------------------------------------")
    print("TP INTEGRADOR PROGRAMACIÓN I - BUSQUEDA Y ORDENAMIENTO")
    print("-----------------------------------------")
    print("1. Busqueda")
    print("2. Ordenamiento")
    print("3. Salir")
    opcion = input("Seleccione una opción: ")

    if opcion == "1":
        dato = busqueda(estudiantes, lista_ordenada)
        if dato is not None:
            print(f"Estudiante encontrado:\nNombre: {dato[0]}, Nota: {
dato[1]}")
            input("Presione Enter para continuar...")
        else:
            print("Estudiante no encontrado.")
            input("Presione Enter para continuar...")
    elif opcion == "2":
        lista_ordenada = ordenar(estudiantes, lista_ordenada)
        input("Presione Enter para continuar...")
    elif opcion == "3":
        print("Hasta luego!")
        break
    else:
        print("Opción no válida. Intente nuevamente.")
```

6. Al principio de la ejecución muestra el menú principal

```
-----------------------------------------
TP INTEGRADOR PROGRAMACIÓN I - BUSQUEDA Y ORDENAMIENTO
-----------------------------------------
1. Busqueda
2. Ordenamiento
3. Salir
Seleccione una opción:
```

7. Al realizarse una búsqueda en primera instancia, la aplicación toma la lista como desordenada, haciendo una búsqueda secuencial.

```
----------------------------------------
TP INTEGRADOR PROGRAMACIÓN I - BUSQUEDA Y ORDENAMIENTO
----------------------------------------
1. Busqueda
2. Ordenamiento
3. Salir
Seleccione una opción: 1
Ingrese el nombre del estudiante a buscar: Emilia
Demora: 0.0000417000 segundos
Estudiante encontrado:
Nombre: Emilia, Nota: 8.3
Presione Enter para continuar...
```

8. En caso de seleccionar “Ordenamiento” te muestra un submenú donde puedes seleccionar uno de los dos métodos disponibles.

```
Seleccione el método de ordenamiento:
1. Descendente (Burbuja)
2. Ascendente (Quicksort)
3. Atrás
Ingrese su opción:
```

9. Una vez finalizado el ordenamiento seleccionado, muestra el tiempo demorado.

```
Seleccione el método de ordenamiento:
1. Ascendente (Burbuja)
2. Descendente (Quicksort)
3. Atrás
Ingrese su opción: 1
Demora: 10.8601680000 segundos
Presione Enter para continuar...
```

```
Seleccione el método de ordenamiento:
1. Ascendente (Burbuja)
2. Descendente (Quicksort)
3. Atrás
Ingrese su opción: 2
Demora: 0.0165608000 segundos
Presione Enter para continuar...
```

10. Por último, podemos realizar una nueva búsqueda. Al tener la lista ordenada, se implementará el método de búsqueda binaria.

```
----------------------------------------
TP INTEGRADOR PROGRAMACIÓN I - BUSQUEDA Y ORDENAMIENTO
----------------------------------------
1. Busqueda
2. Ordenamiento
3. Salir
Seleccione una opción: 1
Ingrese el nombre del estudiante a buscar: Emilia
Demora: 0.0000296000 segundos
Estudiante no encontrado.
Presione Enter para continuar...
```

Metodología utilizada

- Modularización
- Utilización de métodos de búsqueda y ordenamiento
- Utilización de librerías como random, os y time

Resultados obtenidos

Al ejecutar el programa, se observaron los siguientes resultados:

- El algoritmo de ordenamiento quicksort demostró ser significativamente más rápido que el algoritmo de burbuja, especialmente en listas grandes.

- En cuanto a las búsquedas, el algoritmo binario fue mucho más eficiente que la búsqueda secuencial, pero solo cuando la lista estaba previamente ordenada.

- La búsqueda secuencial, aunque más lenta, resultó útil cuando la lista estaba desordenada, ya que no requiere orden previo.

- Se midieron tiempos de ejecución con la función time de Python, lo que permitió comparar empíricamente el rendimiento de cada algoritmo.

Conclusiones

A través de este trabajo, se comprobó que la elección del algoritmo de búsqueda u ordenamiento adecuado puede tener un impacto significativo en el rendimiento del programa. El quicksort y la búsqueda binaria son opciones eficientes cuando se trabaja con grandes volúmenes de datos ordenados, mientras que el método de burbuja o la búsqueda secuencial, aunque más simples, resultan menos eficientes.

Además, el desarrollo de este proyecto permitió afianzar los conceptos teóricos vistos en clase mediante su implementación práctica en Python. Se fortalecieron habilidades de programación modular, medición de rendimiento y análisis comparativo de algoritmos.

Bibliografía

- Apunte de cátedra: Búsqueda y Ordenamiento en Programación.
- https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python
- https://www.alegsa.com.ar/Dic/algoritmo_de_ordenamiento_selection_sort.php
- https://asimov.cloud/blog/programacion-5/que-es-el-algoritmo-de-merge-sort-270
- Artículo en Medium: Comparación entre tipos de búsqueda: https://medium.com/@Emmitta/b%C3%BAsqueda-binaria-c6187323cd72
-

Anexos

- Link al repositorio: https://github.com/scaiciia/IntegProgramacionI