Trabajo Integrador

Algoritmos de Búsqueda y Ordenamiento

Alumno: Vicente Matias López – lopezvicentematias7@gmail.com

Materia: Programación I

Profesor: Julieta Trapé

Comisión: 3

Índice

Introducción

En el desarrollo de programas eficientes, la forma en que se almacenan, recuperan y organizan los datos es fundamental. Es aquí donde entran en juego los algoritmos de búsqueda y ordenamiento, herramientas esenciales en la programación que permiten mejorar el rendimiento de las aplicaciones y optimizar el uso de recursos.

Marco Teórico

En el lenguaje Python, estos algoritmos no solo se pueden aplicar mediante funciones incorporadas como sort() o index(), sino también mediante la implementación personalizada de estructuras como la búsqueda lineal, búsqueda binaria, ordenamiento por burbuja (*bubble sort*), por inserción, por selección y otros métodos más avanzados como *merge sort* o *quick sort*.

Comprender su lógica, comportamiento, y eficiencia mediante el análisis de su complejidad computacional permite al programador tomar decisiones más informadas al enfrentar grandes volúmenes de datos o sistemas en tiempo real. Este estudio abordará la teoría y práctica detrás de estos algoritmos, con especial énfasis en su implementación y análisis dentro del entorno de Python.

Los algoritmos de ordenamiento permiten reorganizar una colección de elementos de acuerdo a un criterio específico (mayor a menor, alfabéticamente, etc.). Entre los más comunes se encuentran: - Bubble Sort: compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. - Insertion Sort: construye una lista ordenada tomando elementos uno a uno e insertándolos en la posición adecuada. - Selection Sort: selecciona el elemento más pequeño del arreglo y lo coloca en su posición final. Por otro lado, los algoritmos de búsqueda tienen como objetivo encontrar un valor dentro de una colección. Destacamos: - Búsqueda lineal: recorre secuencialmente todos los elementos hasta encontrar el deseado. - Búsqueda binaria: eficiente en listas ordenadas, divide el espacio de búsqueda en mitades sucesivas. La búsqueda binaria funciona de la siguiente manera: 1. Se compara el elemento central con el valor buscado. 2. Si son iguales, se retorna la posición. 3. Si el valor es menor, se repite la búsqueda en la mitad izquierda. 4. Si es mayor, en la mitad derecha.

Casos Práctico

Algoritmos de búsqueda

1 – Búsqueda lineal; complejidad: O(n)

Este programa recibe una lista y un elemento a buscar, luego revisa elemento por elemento de dicha lista pasada terminar con su búsqueda. Si encuentra una coincidencia la retorna, sino retornara un -1

```
def searchItem (list, item):
    for i in range(len(list)):
        if list[i] == item:
            return list[i]
    return -1
```

2 – Busqueda binaria; complejidad O(log n)

Solo funciona con listas ordenadas. Divide el rango de búsqueda en mitades.

```
def searchBinary(list, item):
    newList = sorted(list)
    start = 0
    fin = len(newList) - 1
    while start <= fin:
        medio = (start + fin) // 2
        if newList[medio] == item:
            return medio
        elif newList[medio] < item:
            start = medio + 1
        else:
            fin = medio - 1
    return -1
```

Algoritmos de ordenamiento

1 – Ordenamiento por burbuja, complejidad O(n2)**

Compara elementos adyacentes e intercambia si están en el orden incorrecto.

```
def bubleSort(list):
    for i in range(len(list)):
        for j in range(0, len(list) - i - 1):
            if list[j] > list[j + 1]:
                list[j], list[j + 1] = list[j + 1], list[j]
    return list
```

Mejor caso: O(n)

Peor caso: O(n**2)

Promedio: O(n**2)

2 – Ordenamiento por inserción

Inserta cada elemento en su posición correcta dentro de la parte ya ordenada.

```
def insertionSort(list):
    for i in range(1, len(list)):
        key = list[i]
        j = i - 1
        while j >= 0 and key < list[j]:
            list[j+1] = list[j]
            j -= 1
        list[j+1] = key
```

Mejor caso: O(n)

Peor caso: O(n**2)

Promedio: O(n**2)

3 - Ordenamiento de selección

Encuentra el mínimo y lo coloca al frente. Repite para los siguientes.

```
def selectionSort(list):
    n = len(list)
```

```
  for i in range(n):
    minIndex = i
    for j in range(i+1, n):
      if list[j]< list[minIndex]:
        minIndex = j
    list[i], list[minIndex] = list[minIndex], list[i]
```

Mejor caso: O(n**2)

Peor caso: O(n**2)

Promedio: O(n**2)

4 – Ordenamiento por mezcla, mergeSort; complejidad O(n log n)

Divide y conquista: divide la lista en mitades, ordena y fusiona.

```
def mergeSsort(lista):
  if len(lista) > 1:
    medio = len(lista) // 2
    izquierda = lista[:medio]
    derecha = lista[medio:]

    merge_sort(izquierda)
    merge_sort(derecha)

    i = j = k = 0

    while i < len(izquierda) and j < len(derecha):
      if izquierda[i] < derecha[j]:
        lista[k] = izquierda[i]
        i += 1
      else:
        lista[k] = derecha[j]
        j += 1
```

```
        k += 1

    while i < len(izquierda):
        lista[k] = izquierda[i]
        i += 1
        k += 1

    while j < len(derecha):
        lista[k] = derecha[j]
        j += 1
        k += 1
```

Mejor caso: O(n log n)

Peor caso: O(n log n)

Promedio: O(n log n)

5 - Ordenamiento rápido

Elige un pivote y divide en menores y mayores; repite recursivamente.

```
def quickSort(list):
    if len(list) <= 1:
        return list
    else:
        pivot = list[0]
        less = [x for x in list[1:] if x <= pivot]
        greater = [x for x in list[1:] if x > pivot]
        return quickSort(less) + [pivot] + quickSort(greater)
```

Mejor caso: O(n log n)

Peor caso: O(n**2) Si esta ordenado

Promedio: O(n log n)

Comparativa General

- **Eficiencia**: Merge y Quick Sort son mucho más rápidos en listas grandes. Insertion y Bubble son lentos para grandes cantidades de datos.
- **Memoria**: Merge necesita memoria adicional; los demás funcionan in-place (usan el mismo arreglo).
- **Estabilidad**: Merge, Bubble e Insertion conservan el orden relativo de elementos iguales, lo cual es útil si se ordenan objetos con varios atributos.

Escenario	Algoritmo recomendado	Razón
Lista pequeña o casi ordenada	Insertion sort	Simplicidad y buena eficiencia en casos casi ordenados
Lista muy grande sin restricciones de memoria	Merge sort	Rendimiento estable y buena eficiencia
Lista grande con poca memoria	Quick sort	Muy eficiente in-place, aunque sensible a la elección del pivote
Requiere orden estable	Merge sort o insertion sort	Mantienen orden relativo entre elementos iguales
Código muy simple o para enseñanza	Buble sort o selection sort	Fáciles de entender aunque poco eficientes

Aplicaciones en el Mundo Real

- **Quick Sort**: Ampliamente utilizado en bibliotecas estándar como C++, Java y Python por su eficiencia en la práctica.
- **Merge Sort**: Ideal para ordenar archivos grandes almacenados en disco (como en bases de datos) porque accede secuencialmente.
- **Insertion Sort**: Excelente para listas cortas o listas que se actualizan constantemente (como en editores de texto en línea).
- **Bubble/Selection**: Aunque rara vez se usan en producción, son muy útiles para enseñar conceptos básicos de algoritmos.

Metodología Utilizada

Para la elaboración de este trabajo se siguió una metodología de análisis comparativo y estudio documental, con el objetivo de comprender y evaluar diversos algoritmos de búsqueda y ordenamiento. A continuación, se detallan los pasos seguidos:

- **Recolección de información teórica**.
- **Selección de algoritmos relevantes**.
- **Análisis de complejidad**.
- **Comparación estructurada**.
- **Aplicación contextual**.
- **Síntesis y reflexión**.
- **Pruebas en Python con listas de diferentes tamaños**.

Resultados Obtenidos

- Se exploraron distintos **algoritmos de búsqueda**, destacando la **búsqueda lineal** y la **búsqueda binaria**, cada uno con sus ventajas según la estructura y el orden de los datos.
- Se analizaron en detalle **cinco algoritmos de ordenamiento** (Bubble Sort, Insertion Sort, Selection Sort, Merge Sort y Quick Sort), comparando su eficiencia, estabilidad, uso de memoria y casos de aplicación.
- A través del análisis de complejidad, se determinaron los **escenarios ideales para cada algoritmo**, considerando su rendimiento en el mejor, peor y caso promedio.
- Se establecieron **criterios de selección** entre algoritmos, según el tamaño de la lista, si está previamente ordenada, la necesidad de ordenamiento estable y las restricciones de memoria.
- Se presentaron ejemplos concretos del uso de estos algoritmos en entornos reales como motores de búsqueda, editores de texto, sistemas de bases de datos y bibliotecas de programación modernas.
- Se concluyó que la **selección adecuada de un algoritmo depende del contexto específico del problema**, y que conocer las fortalezas y debilidades de cada uno es clave para desarrollar software eficiente

Conclusiones

Los algoritmos de búsqueda y ordenamiento representan herramientas fundamentales dentro del campo de la informática y la programación. A lo largo de este trabajo, se ha analizado en profundidad su funcionamiento, eficiencia, ventajas y desventajas, así como su relevancia práctica en distintos contextos.

Comprender cómo y cuándo aplicar cada algoritmo permite no solo optimizar el rendimiento de los programas, sino también fortalecer el pensamiento lógico y la capacidad de resolución de problemas. Saber elegir entre una búsqueda lineal o binaria, o entre un ordenamiento por inserción o uno por mezcla, puede marcar una diferencia significativa en la eficiencia de un sistema, especialmente cuando se trabaja con grandes volúmenes de datos o recursos limitados.

En conclusión, los algoritmos estudiados no son solo estructuras teóricas: son herramientas vivas que impulsan desde tareas cotidianas, como ordenar contactos en un celular, hasta sistemas complejos como los motores de búsqueda y bases de datos globales. Su estudio y dominio son esenciales para cualquier persona que aspire a desarrollarse profesionalmente en el ámbito tecnológico y científico

Bibliografía

https://github.com/Asmilex/DAI/blob/6f31e1fcd58b0cadaf3aa4fbd7be6cf55ef7065e/app/ejercicios/ordenacion.py

https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/

https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

https://es.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/sorting

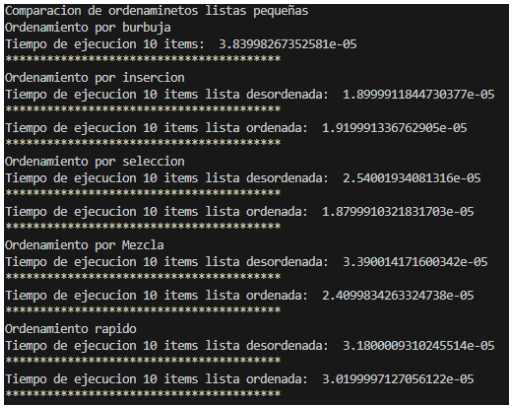
https://es.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort

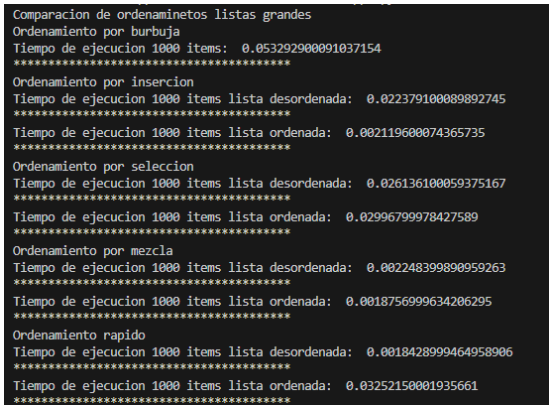
https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

https://es.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort

Anexos

- Captura de pantalla del programa funcionando.





- Repositorio en GitHub: https://github.com/lopezMatiasV/integradorN1UTN

- Video explicativo: https://youtu.be/FecPP2xI-2k