Trabajo Integrador: Algoritmos de Búsqueda y Ordenamiento en Python

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Materia:

Materia: Programación I

Alumnos:

Cufré, Facundo (Comisión 12) - facundo.cufre@tupad.utn.edu.ar
Sánchez, Noelia (Comisión 21) - noelia.sanchez@tupad.utn.edu.ar

Docentes:

Prof. Quirós, Nicolás
Prof. Enferrel, Ariel

Tutores:

Bianchi, Neyen
Chiroli, Luciano

Fecha de Entrega:

9 de Junio de 2025

Contenido

TRABAJO INTEGRADOR : ALGORITMO DE BÚSQUEDA Y ORDENAMIENTO EN PYTHON 3
Introducción 3
Marco Teórico 4
Caso Práctico 6
Imagen 1: Cuadro comparativo de tiempos según tipos de búsqueda y de ordenamiento 8
Metodología Utilizada 9
Resultados Obtenidos 9
Conclusiones 10
Bibliografía 11
Anexos 11

TRABAJO INTEGRADOR : ALGORITMO DE BÚSQUEDA Y ORDENAMIENTO EN PYTHON

Introducción

En este trabajo se desarrollará el tema de Búsqueda y Ordenamiento, donde la búsqueda se refiere a encontrar datos dentro de una colección, y el ordenamiento a organizarlos en un orden específico. Se evaluarán dos tipos de búsqueda: lineal, y binaria, analizando su comportamiento en relación con el tamaño de la muestra con la y el tiempo que tardan en encontrar un elemento. También se analizarán dos métodos de ordenamiento, Bubble Sort (ordenamiento por burbuja) y Quick Sort (ordenamiento rápido).

Conociendo el comportamiento de estos algoritmos, es posible elegir el más adecuado según el caso, priorizando la eficiencia, lo cual impacta en el rendimiento del sistema y en la experiencia del usuario.

Para este trabajo, se utiliza una lista de apellidos de alumnos como ejemplo de colección pequeña, la cual será ordenada alfabéticamente según los métodos mencionados. En el caso de listas grandes, se generará una de diez mil números aleatorios a modo de comparación. Finalmente se realiza la búsqueda en ambas listas permitiendo al usuario ingresar un valor y encontrar su posición.

El objetivo principal de este trabajo integrador es comprender los distintos algoritmos de búsquedas y ordenamiento, evaluando sus ventajas, desventajas y los contextos en los que se puede aprovechar mejor a cada uno. Para esto se desarrolló un pequeño programa que muestra por consola los tiempos de ejecución de las búsquedas y ordenamientos, lo que permite comparar su eficiencia de forma práctica.

Marco Teórico

La búsqueda y ordenamiento son un tema central dentro de la programación ya que se utilizan para encontrar y organizar datos, lo que es esencial para optimizar el rendimiento de las aplicaciones.

Búsqueda: Es la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento. Son herramientas esenciales en la ciencia de la computación usados para localizar elementos específicos dentro de una colección de datos

Tipos de Búsqueda:

- **Búsqueda Lineal:** se usa para un arreglo desordenado. Principalmente hace comparaciones una por una del elemento que se va a buscar con los del arreglo. Toma tiempo lineal, es decir O(n).
- **Búsqueda Binaria:** Es utilizado para un arreglo ordenado. Compara principalmente el elemento del medio del arreglo y si este es igual al valor buscado, lo devuelve. De lo contrario, busca en la mitad izquierda o derecha según el resultado de la comparación (Si el elemento del medio es menor o mayor). El algoritmo es más rápido que el linear y toma un tiempo O(Log n).
- **Búsqueda de interpolación**: La búsqueda por interpolación (Interpolation Search) es una mejora de la búsqueda binaria para casos en los que los valores de un arreglo ordenado están distribuidos de forma uniforme. La interpolación construye nuevos puntos de datos dentro del rango de un conjunto discreto de puntos conocidos.
- **Búsqueda de hash:** es una técnica que permite una búsqueda rápida y eficiente en grandes conjuntos de datos, sin la necesidad de que los datos estén ordenados previamente. Se utiliza una función hash para mapear los datos a una tabla de hash, donde se almacenan. La función hash genera un índice único para cada dato, lo que facilita su ubicación rápida en la tabla.

Ordenamiento: Es la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento.

El ordenamiento se efectúa con base en el valor de algún campo en un registro.El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado.

Ej:dado un arreglo [10, 20, 5, 2] se transforma en [2, 5, 10, 20] después del ordenamiento en orden decreciente.

Existen diferentes tipos de ordenamiento para distintos tipos de entradas. Por ejemplo, un arreglo binario, un arreglo de caracteres, un arreglo con un rango amplio de valores, un arreglo con muchos duplicados , o arreglos pequeños o grandes.

Tipos de Ordenamiento:

- **Selection Sort**: Es un algoritmo de ordenamiento basado en comparaciones. Ordena un arreglo seleccionando repetidamente el elemento más pequeño (o más grande) de la porción no ordenada y lo intercambia con el elemento de esa porción. Este proceso continúa hasta que todo el arreglo esté ordenado.
- **Bubble Sort**: es el algoritmo de ordenamiento más simple. Funciona intercambiando repetidamente los elementos adyacentes si están en el orden incorrecto. Este algoritmo no es adecuado para conjuntos de datos grandes ya que su complejidad temporal promedio y en el peor caso son bastante alta.
- **Insertion Sort**: es un algoritmo de ordenamiento simple que funciona de forma iterativa, insertando cada elemento de una lista desordenada en su posición correcta dentro de la porción ordenada de la lista. Es como ordenar cartas en la mano: se dividen en dos grupos, las cartas ordenadas y las desordenadas. Luego, se toma una carta del grupo desordenado y se la coloca en el lugar correcto dentro del grupo ordenado.
- **Quick Sort**: algoritmo de ordenamiento basado en el principio de “Divide y Reinarás”. Selecciona un elemento como “pivot” y divide el arreglo en particiones alrededor de este pivot, colocándolo en su posición correcta dentro del arreglo ordenado. De esta forma, se reduce el problema en pequeños subproblemas.

Caso Práctico

Para el caso práctico se decidió utilizar dos algoritmos de búsqueda y dos de ordenamiento.

Los algoritmos de búsqueda utilizados fueron:

- Búsqueda Lineal
- Búsqueda Binaria

Por otro lado, los algoritmos de ordenamiento que usamos fueron:

- Bubble Sort
- Quick Sort

Función burbuja (Bubble Sort):

```python
# Ordenamiento burbuja
def burbuja(lista):
    # Se crea una copia de la lista para no afectar la original
    lista = lista.copy()
    # Se obtiene el largo de la lista
    largo_lista = len(lista)
    # Bucle externo que recorre toda la lista
    for i in range(largo_lista):
    # Bucle interno que comparará elementos adyacentes y los intercambia si están desordenados
        for j in range(0, largo_lista - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista
```

Función quicksort:

```python
# Quick Sort
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        # Se elige el primer elemento como pivot
        pivot = lista[0]
        # Sublista con los elementos que sean menores o iguales al pivot
        menos = [x for x in lista[1:] if x <= pivot]
        # Sublista con los elementos que sean mayores al pivot
        mayor_que = [x for x in lista[1:] if x > pivot]
        # Se llama recursivamente a la función quicksort y se le concatenan los resultados
        return quicksort(menos) + [pivot] + quicksort(mayor_que)
```

Función busqueda_binaria:

```python
# Búsqueda binaria:
def busqueda_binaria(lista, objetivo):
    # Se definen los extremos
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        # Se calcula el índice del elemento del medio
        medio = (izquierda + derecha) // 2
        # Si el elemento del medio es el objetivo, se retorna su índice
        if lista[medio] == objetivo:
            return medio
        # Si el objetivo es mayor, se descarta la mitad izquierda
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        # Si el objetivo es menor, se descarta la mitad derecha
        else:
            derecha = medio - 1
    return -1
```

Función busqueda_lineal:

```python
# Búsqueda lineal:
def busqueda_lineal(lista, objetivo):
    # Se recorre la lista
    for i in range(len(lista)):
    # Si el elemento actual es igual al objetivo retorna el índice
        if lista[i] == objetivo:
            return i
    # Si no encuentra el objetivo retorna -1
    return -1
```

Utilizando estos algoritmos, se ejecutaron y se pudo capturar los tiempos de cada uno de ellos.

En el siguiente cuadro se muestran los tiempos obtenidos para cada algoritmo luego de la aplicación del código:

Tiempos		
Búsqueda		
	Lista grande	**Lista pequeña**
Búsqueda lineal	0,000384	9926 x e-06
Búsqueda binaria	917969 e -05	86049 e -06
Ordenamiento		
	Lista grande	**Lista pequeña**
Bubble Sort	7,317361	0,0001861
Quick Sort	0,033102	0,0001184

Imagen 1: Cuadro comparativo de tiempos según tipos de búsqueda y de ordenamiento

Se eligieron los algoritmos de ordenamiento bubble sort y quick sort ya que estos nos brindan un contraste entre dos enfoques muy distintos: por un lado bubble sort es sencillo de entender e implementar, pero poco eficiente en términos de rendimiento. Por otro lado, quick sort representa una forma mucho más rápida y eficiente de ordenar, pero como contrapartida tiene una lógica más difícil de implementar que la bubble sort. Esta comparación permite apreciar cómo el diseño de algoritmo impacta directamente en el tiempo de ejecución, especialmente con listas grandes.

Por otro lado elegimos los algoritmos de búsqueda lineal y binaria ya que representan dos formas muy diferentes de resolver el mismo problema. La lineal es más simple y no requiere que la lista esté ordenada, lo que la hace ideal para listas pequeñas o no estructuradas. Por otro lado, la binaria es mucho más eficiente, pero necesita tener previamente ordenada la lista. Comparar ambos permitió entender que el contexto de tener la lista ordenada, o no, así como también el tamaño de la lista influye directamente en la

elección del algoritmo más adecuado.

Metodología Utilizada

El trabajo práctico integrador se realizó con los siguientes criterios:

- Recopilación de teoría de diferentes fuentes.
- Estudio de los conceptos para luego realizar un programa en Python que implementa los algoritmos estudiados.
- Comparación de los métodos de búsqueda y ordenamiento para tener una visión más clara de su eficacia.
- Registro de resultados, imprimiéndolos por consola, demostrando la funcionalidad del código.
- Preparación y presentación de diapositivas que resumen brevemente el tema.
- Realización de un video explicativo en el que se aborda el código realizado en Python.
- Finalmente, recopilación de toda la documentación, tanto teórica como práctica, en este documento.

Resultados Obtenidos

- Utilizando la función `burbuja`, se ordenó con éxito la lista `apellidos`.
- Se pudo registrar exitosamente el tiempo en ordenar tanto para el Bubble Sort como el Quick Sort, en donde se pudo notar la diferencia abismal de rendimiento para los casos de listas grandes, donde la Bubble Sort demoró 7.32 segundos.
- El menú para los algoritmos de búsqueda funcionó correctamente y devolvió los tiempos de la búsqueda lineal y la binaria, concluyendo que para listas pequeñas la diferencia es prácticamente imperceptible, pero para los casos de listas grandes la búsqueda binaria es ampliamente más rápida que la lineal.
- Se comprendió la complejidad y funcionamiento de cada algoritmo.

Conclusiones

Como conclusión, pudimos observar que respecto a los tipos de ordenamiento, el quick sort es mucho más eficaz y rápido que el bubble sort, salvo para casos de listas muy pequeñas, por lo cual en la mayoría de los casos el bubble sort queda obsoleto. En nuestro proyecto, visualizamos que para listas pequeñas el Quick Sort fue más rápido, y en el caso de listas grandes la diferencia fue incluso mayor: bubble sort tardó unos 7.32 segundos en ordenar una lista de solo diez mil elementos, número que resulta extremadamente alto.

En cuanto a las búsquedas, para listas pequeñas como la de apellidos, la diferencia entre búsqueda lineal y binaria fue prácticamente imperceptible. Sin embargo, en listas grandes , la búsqueda binaria demostró ser mucho más rápida. La diferencia principal entre ambos métodos es que la búsqueda binaria requiere que la lista esté previamente ordenada, aunque en nuestro proyecto utilizamos la lista ordenada para ambos casos, por un tema de practicidad y además porque el que esté ordenada o no no hace variar el tiempo de respuesta para la búsqueda lineal.

Esto demuestra la importancia de conocer las características de cada algoritmo y su aplicabilidad según el tamaño y estado de los datos.

Bibliografía

- Monografías: https://www.monografias.com/trabajos/algordenam/algordenam
- Geeks for Geeks: https://www.geeksforgeeks.org
- Wikipedia: https://es.wikipedia.org/wiki/Tabla_hash

Anexos

- Repositorio de GitHub con toda la documentación requerida
- Video explicativo: https://youtu.be/eLkBkrRqE3A