Trabajo Práctico Integrador

“Algoritmos de Búsqueda y Ordenamiento en Python”

Tecnicatura Universitaria en Programación a Distancia – UTN

Año 2025

Alumnos: Gabriel Carbajal – Ignacio Carné

Materia: Programación I

Profesor: Ariel Enferrel

Fecha de entrega: 8/6/2025

Link al video en Youtube:

Índice

1. Introducción

El trabajo integrador se basa en la investigación, análisis e implementación de algoritmos de búsqueda y ordenamiento de datos y la implementación de hará en el lenguaje de programación Pyhton 3.

La elección de este tema es debida a que estos algoritmos son utilizados de manera ubicua y son herramientas fundamentales para los programadores a la hora de trabajar con datos.

Se desarrollarán algoritmos de búsqueda y ordenamiento y se compararán los pros y contras en cada caso.

2. Marco Teórico

Los algoritmos de búsqueda son un conjunto de pasos, escritos en algún lenguaje de programación, que nos ayuda a encontrar un dato (si es que existe) dentro de un conjunto de datos y nos dice en qué posición se encuentra dentro del conjunto.

Los algoritmos de búsqueda se encuentran presentes en casi todos los sistemas por nosotros utilizados; desde motores de búsqueda como sitios web o bases de datos hasta plataformas de e-commerce, redes sociales, servicios de streaming, sistemas de archivos e inteligencia artificial (Ďurišinová, 2023).

Comprender los diferentes algoritmos de búsqueda y cómo utilizarlos, puede mejorar el rendimiento y la eficiencia de los programas y así mejorar la experiencia de usuario.

Destacamos dos algoritmos de búsqueda en este trabajo:

- Búsqueda lineal: o búsqueda secuencial, itera sobe cada elemento del conjunto, comparando este con el elemento objetivo (el que buscamos) hasta que encuentre una coincidencia. Sencilla pero lenta.
- Búsqueda binaria iterativa: para que funcione el algoritmo, los datos en el conjunto debe estar ordenados. Divide el conjunto de datos en mitades de manera sucesiva, determinando en cada mitad si el elemento es igual, mayor o menor que el objetivo. Es rápido, especialmente con conjuntos grandes.
- Búsqueda binaria recursiva: opera de la misma manera que la búsqueda binaria iterativa pero no usa un bucle para hacer las sucesivas divisiones del espacio sino que la función se llama a sí misma con nuevos valores. Es igual de eficiente pero puede sufrir sobrecarga de la pila de llamadas (Chinni, 2022).

Un algoritmo de ordenamiento organiza una colección de elementos (números, palabras, objetos, etc.) siguiendo un orden específico. Algunos de los más conocidos son:

- Bubble Sort: Recorre repetidamente la lista, comparando elementos adyacentes y cambiándolos si están en el orden incorrecto. Es sencillo, pero poco eficiente para listas grandes.

- Insertion Sort: Toma los elementos uno a uno y los va insertando en la posición correcta dentro de una lista ordenada parcial.

- Selection Sort: Encuentra el menor elemento del arreglo y lo coloca en su posición correcta, repitiendo el proceso con el resto.

- Quick Sort (más avanzado): Divide y conquista; selecciona un pivote y ordena recursivamente los subconjuntos menores y mayores.

- Merge Sort: También basado en dividir y conquistar; divide el arreglo en mitades, las ordena por separado y luego las fusiona.

La notación Big O es una forma de describir la eficiencia de un algoritmo. Nos indica qué tan rápido se vuelve más deficiente el algoritmo a medida que el problema se hace más grande.

- Bubble Sort: $O(n^2)$

- Insertion Sort: $O(n^2)$

- Selection Sort: $O(n^2)$

- Merge Sort: $O(n \log n)$

- Quick Sort: $O(n^2)$, promedio $O(n \log n)$

3. Caso Práctico

Algoritmos de búsqueda en Python 3

Búsqueda lineal:

```
11  # Algoritmo de búsqueda lineal
12  def busqueda_lineal(lista, objetivo):
13      for i in range(len(lista)):
14          if lista[i] == objetivo:
15              return i
16      return "El objetivo no está en la lista"
```

Búsqueda binaria iterativa:

```python
# Algoritmo de búsqueda binaria iterativa
def busqueda_iterativa(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return "El objetivo no está en la lista"
```

Búsqueda binaria recursiva:

```python
# Búsqueda binaria recursiva
def busqueda_recursiva(lista, objetivo, inicio = 0, fin = None):
    if fin == None:
        return -1

    if inicio > fin:
        return -1

    medio = (inicio + fin) // 2

    if lista[medio] == objetivo:
        return medio
    elif lista[medio] < objetivo:
        return busqueda_recursiva(lista, objetivo, medio + 1, fin)
    else:
        return busqueda_recursiva(lista, objetivo, inicio, medio - 1)
```

<u>Implementación de algoritmos de ordenamiento en Python:</u>

Ordenamiento por burbuja, por inserción y por selección:

Programacion - TP INTEGRADOR Gabriel Carbajal.py ●

C: > Users > gcarbajal > Desktop > Gabi > UTN Programacion > Programacion - TP INTEGRADOR Gabriel Carbajal.p

```python
#BUBBLE SORT
#Compara elementos de a pares y los intercambia si están en el orden incorrecto.
#Los números grandes van “flotando” al final, por eso el nombre "burbujas"

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

```python
#INSERTION SORT
#Se asemeja a un mazo de cartas
#Va insertando los numeros en la posición correcta de los anteriores ya ordenados.

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >=0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

#SELECTION SORT
#Busca el número más chico y lo pone al principio, luego repite lo mismo con el resto

def selection_sort(arr):
    n = len(arr)
    for i in range(n): # Recorremos toda la lista desde el primer hasta el último elemento
        min_idx = i #min_idx es la posicion donde está el valor minimo
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Hacemos una lista random para probar el ordenamiento
datos = [64, 25, 12, 22, 11]

# Copiamos las listas para cada prueba
lista_bubble = datos[:]
lista_insertion = datos[:]
lista_selection = datos[:]

#Ordenamos la lista con el algoritmo correspondiente
bubble_sort(lista_bubble)
insertion_sort(lista_insertion)
selection_sort(lista_selection)

#Mostramos el resultado en pantalla
print("Bubble Sort:", lista_bubble)
print("Insertion Sort:", lista_insertion)
print("Selection Sort:", lista_selection)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\gcarbajal> & C:/Users/gcarbajal/AppData/Local/Programs/Python/I
EGRADOR Gabriel Carbajal.py"
Bubble Sort: [11, 12, 22, 25, 64]
Insertion Sort: [11, 12, 22, 25, 64]
Selection Sort: [11, 12, 22, 25, 64]
PS C:\Users\gcarbajal>
```

4. Metodología Utilizada

Para la elaboración del trabajo integrador nos basamos en la información puesta a disposición por la cátedra en el aula virtual y búsquedas en internet.

Se utilizó Python 3 para escribir y ejecutar los algoritmos. Se realizaron pruebas de los algoritmos con diferentes conjuntos de datos.

Para comparar los tres algoritmos de ordenamiento (Bubble Sort, Insertion Sort y Selection Sort), se utilizó el mismo conjunto de datos inicial. Luego se copió esa misma lista tres veces, una para cada algoritmo para asegurar que todos funcionen con los mismos valores y la comparación sea justa. Para fializar se imprimieron los resultados ordenados por cada método para comprobar que los tres lleguen al mismo resultado.

5. Resultados Obtenidos

Tanto los algoritmos de búsqueda como los de ordenamiento funcionaron de manera correcta, con diferentes conjuntos de valores.

Se pudo observar las diferencias de eficiencia entre el algoritmo de búsqueda lineal y el de búsqueda binaria (a través de la medición del tiempo de ejecución, explicado en el video).

6. Conclusiones

Los algoritmos de búsqueda y ordenamiento son esenciales en la vida informática actual, donde con más frecuencia se realizan búsquedas en internet, en bases de datos, redes sociales, etc.

La búsqueda lineal se sencilla de implementar y no depende del orden de los elementos pero tiene las desventaja de perder eficiencia rápidamente a medida que el número de datos a analizar aumenta. El algoritmo binario es mucho más eficiente, tanto en su implementación iterativa como recursiva, si bien tiene el requisito del ordenamiento previo de los datos.

El conocimiento de los algoritmos estudiados posiciona a los desarrolladores de mejor manera a la hora de elegir que algoritmo utilizar en función del número de datos y requisitos del sistema (ausum.cloud) y de esa decisión depende, en definitiva, la experiencia de usuario y el éxito del producto ofrecido.

7. Bibliografía

Ďurišinová, M. (29 de Mayo de 2023). *Entendiendo los Distintos Tipos de Algoritmos de Búsqueda.* Luigi's box. https://www.luigisbox.es/blog/tipos-de-algoritmos-de-busqueda

Chinni, C. (2 de julio de 2022). *Pila de llamadas de función en C.* Scaler.com https://www.scaler.com/topics/c/c-function-call-stack/

ausum.cloud. (s.f.). https://ausum.cloud/escalabilidad-vertical-vs-horizontal-diferencias-ventajas-kubernetes-y-cual-elegir/

8. Anexos

Link al video en Youtube: https://youtu.be/0GgEkAP_rdU

Link al repositorio github: https://github.com/Tecnac23/TP-Integrador-Programacion1.git