Trabajo Práctico Integrador Programación 1

[Título del TP]

Alumnos

Bustamante Erica - ericaba@gmail.com

Betancort Anahí -anahi.betancort@gmail.com

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación 1

Docente Titular

Prof. Cinthia Rigoni

Docente Tutor

Prof. Martín A. García

Tabla de contenido

Trabajo Práctico Búsqueda y ordenamiento

Introducción

En este trabajo analizamos distintos algoritmos de búsqueda y ordenamiento en Python, entendiendo su funcionamiento y cuándo es mejor usar cada uno.

Estudiamos búsqueda lineal y binaria (iterativa y recursiva), probándose en listas de distintos tamaños y condiciones. Luego, extendemos el análisis a los algoritmos de ordenamiento, comparando métodos clásicos como burbuja y selección con la función sorted() de Python.

Para aplicar estos conceptos, creamos un sistema de gestión de empleados que permite agregar, buscar y ordenar datos según distintos criterios, eligiendo los algoritmos más eficientes según nuestras pruebas. Así, combinamos teoría y práctica, evaluando rendimiento y aplicabilidad.

Marco teórico

Algoritmos de Ordenamiento y Búsqueda en Python

En el mundo del desarrollo de software, los algoritmos de búsqueda y ordenamiento juegan un papel fundamental, estas técnicas permiten organizar y obtener datos de una manera muy eficiente, lo que es esencial para optimizar el rendimiento de las

aplicaciones. En este trabajo veremos algunos ejemplos de algoritmos de búsqueda en Python.

¿Qué son los Algoritmos de Búsqueda?

Los algoritmos de búsqueda son métodos que nos permiten encontrar la ubicación de un elemento específico dentro de una lista de elementos. Dependiendo de la lista necesitarás utilizar un algoritmo u otro, por ejemplo si la lista tiene elementos ordenados, puedes usar un algoritmo de búsqueda binaria, pero si la lista contiene los elementos de forma desordenada este algoritmo no te servirá, para buscar un elemento en una lista desordenada deberás utilizar un algoritmo de búsqueda lineal.

Estos algoritmos son dos de los más relevantes y conocidos en la programación, a continuación, veremos ejemplos de estos dos algoritmos.

Búsqueda Lineal

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implica recorrer una lista de elementos uno por uno hasta encontrar un elemento específico. Este algoritmo es muy sencillo de implementar en código pero puede ser muy ineficiente dependiendo del largo de la lista y la ubicación donde está el elemento.

Ventajas y Desventajas del Algoritmo de Búsqueda Lineal

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

El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias.

Ventajas y Desventajas del Algoritmo de Búsqueda Binaria

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.

Ordenamiento

El ordenamiento es el proceso de reorganizar un conjunto de datos según un criterio específico, como orden alfabético o numérico ascendente. Esta operación es clave porque mejora la eficiencia de otras tareas, como la búsqueda, el análisis o la fusión de datos.

Los algoritmos de ordenamiento permiten estructurar la información de forma lógica, facilitando la interpretación y el procesamiento de datos. Algunos de los algoritmos más conocidos son:

- Ordenamiento por burbuja (Bubble Sort): Compara elementos adyacentes y los intercambia si están desordenados. Es simple, pero poco eficiente ($O(n^2)$).

- Ordenamiento por selección (Selection Sort): Encuentra el menor elemento y lo coloca al inicio, repitiendo el proceso. También tiene una complejidad de $O(n^2)$.

- Ordenamiento por inserción (Insertion Sort): Inserta cada nuevo elemento en la posición adecuada dentro de una lista parcialmente ordenada. Es útil para listas pequeñas o casi ordenadas.

- Ordenamiento rápido (Quick Sort): Utiliza una estrategia de "divide y vencerás", seleccionando un pivote para particionar la lista. Su rendimiento promedio es O(n log n), siendo muy eficiente en la práctica.

- Ordenamiento por mezcla (Merge Sort): Divide el conjunto en partes, las ordena recursivamente y luego las fusiona. También tiene una complejidad de O(n log n), con un comportamiento más predecible que Quick Sort.

El ordenamiento no solo mejora la eficiencia de la búsqueda binaria, sino que también facilita tareas como la detección de duplicados, el análisis de tendencias o la visualización estructurada de datos.

Importancia y Aplicaciones

La combinación de algoritmos de búsqueda y ordenamiento permite optimizar significativamente el rendimiento de un programa. Algunas de sus ventajas más destacadas son:

- Eficiencia: Reducen los tiempos de ejecución, especialmente en aplicaciones que manipulan grandes volúmenes de información.

- Organización: Permiten estructurar los datos de forma lógica y coherente, facilitando su análisis.

- Escalabilidad: Se adaptan a distintos tamaños de entrada, desde pequeños conjuntos hasta grandes bases de datos.

- Precisión: Aseguran resultados exactos y relevantes, reduciendo errores.

- Versatilidad: Se aplican en numerosos contextos, como sistemas de gestión, motores de búsqueda, análisis de datos, redes, videojuegos, entre otros.

Caso práctico

El objetivo de este trabajo fue analizar y comparar la eficiencia de distintos algoritmos de búsqueda y ordenamiento en Python, aplicando luego estos conocimientos en un caso práctico: un sistema de gestión de empleados.

Primero, evaluamos el rendimiento de tres algoritmos de búsqueda: lineal, binaria iterativa y binaria recursiva. Luego, ampliamos el análisis incorporando algoritmos de ordenamiento, esenciales para mejorar la eficiencia de ciertas búsquedas. Estudiamos burbuja, selección y la función sorted() de Python, basada en Timsort.

Finalmente, desarrollamos un sistema de gestión de empleados donde aplicamos los algoritmos más eficientes según nuestras pruebas, integrando teoría y práctica para optimizar su funcionamiento.

Código fuente

Algoritmos de búsqueda (busquedas.py)

Incluye tres funciones: búsqueda lineal, binaria iterativa y recursiva.

```
def busqueda_binaria_iterativa(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
```

```
    while inicio <= fin:
        medio = (inicio + fin) // 2

        if lista[medio] == objetivo:
            return medio  # Encontró el valor, retorna índice
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1

    return None  # No se encontró el valor



def busqueda_binaria_recursiva(lista, objetivo, inicio=0, fin=None):
    if fin is None:
        fin = len(lista) - 1

    if inicio > fin:
        return -1  # Valor no encontrado

    medio = (inicio + fin) // 2

    if lista[medio] == objetivo:
```

```
        return medio  # Se encontró el valor

    elif lista[medio] < objetivo:
        # Buscar en la mitad derecha
        return busqueda_binaria_recursiva(lista, objetivo, medio + 1, fin)
    else:
        # Buscar en la mitad izquierda
        return busqueda_binaria_recursiva(lista, objetivo, inicio, medio - 1)

#función lineal, busqueda lineal
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

Algoritmos de ordenamiento (ordenamientos.py)

Incluye funciones propias de ordenamiento burbuja y selección, además del uso de sorted():

```
# ordenamientos.py

import timeit
import random
```

```
# ------------------- ALGORITMOS DE ORDENAMIENTO -------------------


def ordenamiento_burbuja(lista):
    lista = lista.copy()
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista


def ordenamiento_seleccion(lista):
    lista = lista.copy()
    n = len(lista)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista


def ordenamiento_python(lista):
```

```
    return sorted(lista)


# ------------------- PRUEBA DE RENDIMIENTO --------------------


def prueba_ordenamiento(funcion, lista):
    tiempo = timeit.timeit(lambda: funcion(lista), number=1)
    return tiempo


if __name__ == "__main__":
    tamanos = [100, 500, 1000]
    print("Comparación de tiempos de ordenamiento (en segundos):")
    print("{:<10} {:<15} {:<15} {:<15}".format("Tamaño", "Burbuja", "Selección", "Sorted()"))

    for tamano in tamanos:
        lista = random.sample(range(tamano * 10), tamano)

        t_burbuja = prueba_ordenamiento(ordenamiento_burbuja, lista)
        t_seleccion = prueba_ordenamiento(ordenamiento_seleccion, lista)
        t_sorted = prueba_ordenamiento(ordenamiento_python, lista)

        print("{:<10} {:<15.5f} {:<15.5f} {:<15.5f}".format(
            tamano, t_burbuja, t_seleccion, t_sorted))
```

Generación de listas (generador.py)

Se creó una función para generar listas de números enteros únicos y ordenados aleatoriamente:

```
import random


def generar_lista_ordenada(tamano):
    return sorted(random.sample(range(tamano*10), tamano))
```

Medición de rendimiento (pruebas.py)

Usamos timeit para medir el tiempo que tardan las funciones en ejecutarse:

```
import timeit


def prueba_busqueda(funcion, lista, objetivo):
    # Usamos timeit para medir el tiempo
    tiempo = timeit.timeit(lambda: funcion(lista, objetivo), number=1000)
    return tiempo
```

Ejecución y pruebas (main.py)

Se ejecutaron comparaciones entre los algoritmos de búsqueda utilizando listas de tamaños crecientes:

10, 100, 1.000, 10.000, 100.000.

Se midió cuánto tardaba cada algoritmo en encontrar un valor existente y también se realizaron pruebas con un valor que no está presente en la lista. Finalmente, se incluyó una prueba con una lista pequeña como caso base.

Además, se incorporó la comparación de algoritmos de ordenamiento sobre listas de tamaño 100, 500 y 1.000, evaluando su eficiencia en tiempo de ejecución.

```
# main.py


from busquedas import busqueda_binaria_iterativa, busqueda_binaria_recursiva, busqueda_lineal

from generador import generar_lista_ordenada

from pruebas import prueba_busqueda

from ordenamientos import ordenamiento_burbuja, ordenamiento_seleccion, ordenamiento_python, prueba_ordenamiento

import random


# ------------------- PRUEBAS DE BÚSQUEDA -------------------

tamanos = [10, 100, 1000, 10000, 100000]


print("\nTiempos de ejecución (en milisegundos) para 1000 ejecuciones de búsqueda:")

print("{:<10} {:<15} {:<15} {:<15}".format("Tamaño", "Lineal", "Binaria It.", "Binaria Rec."))


for tamano in tamanos:
```

```
    lista = generar_lista_ordenada(tamano)
    objetivo = random.choice(lista)

    tiempo_lineal = prueba_busqueda(busqueda_lineal, lista, objetivo)
    tiempo_iter = prueba_busqueda(busqueda_binaria_iterativa, lista, objetivo)
    tiempo_rec = prueba_busqueda(busqueda_binaria_recursiva, lista, objetivo)

    print("{:<10} {:<15.5f} {:<15.5f} {:<15.5f}".format(
        tamano, tiempo_lineal * 1000, tiempo_iter * 1000, tiempo_rec * 1000))

# Prueba adicional con objetivo inexistente
print("\nPrueba con objetivo que NO está en la lista:")
objetivo = -1
lista = generar_lista_ordenada(10000)

print("Lineal:", prueba_busqueda(busqueda_lineal, lista, objetivo) * 1000, "ms")
print("Binaria Iterativa:", prueba_busqueda(busqueda_binaria_iterativa, lista, objetivo) * 1000, "ms")
print("Binaria Recursiva:", prueba_busqueda(busqueda_binaria_recursiva, lista, objetivo) * 1000, "ms")

# Prueba con lista pequeña
print("\nResultado con lista pequeña:")
```

```
lista_pequena = [1, 2, 3]

objetivo = 2

print("Resultado búsqueda lineal:", busqueda_lineal(lista_pequena, objetivo))

print("Resultado binaria iterativa:", busqueda_binaria_iterativa(lista_pequena, objetivo))

print("Resultado binaria recursiva:", busqueda_binaria_recursiva(lista_pequena, objetivo))



# ---------- PRUEBAS DE ORDENAMIENTO ----------

print("\nComparación de tiempos de ordenamiento (en segundos):")

print("{:<10} {:<15} {:<15} {:<15}".format("Tamaño", "Burbuja", "Selección", "Sorted()"))



for tamano in [100, 500, 1000]:

    lista = random.sample(range(tamano * 10), tamano)



    t_burbuja = prueba_ordenamiento(ordenamiento_burbuja, lista)

    t_seleccion = prueba_ordenamiento(ordenamiento_seleccion, lista)

    t_sorted = prueba_ordenamiento(ordenamiento_python, lista)



    print("{:<10} {:<15.5f} {:<15.5f} {:<15.5f}".format(

        tamano, t_burbuja, t_seleccion, t_sorted))
```

Ejecución de pruebas

Se realizaron las siguientes pruebas:

- Listas con tamaños: 10, 100, 1.000, 10.000, 100.000.
- Pruebas con objetivo presente y no presente en la lista.
- Comparación de búsquedas: lineal, binaria iterativa y recursiva.
- Comparación de ordenamientos: burbuja, selección y sorted().
- Prueba adicional con lista muy pequeña como caso base.

```
Tiempos de ejecución (en milisegundos) para 1000 ejecuciones de búsqueda:
Tamaño     Lineal          Binaria It.     Binaria Rec.
10         0.47420         0.29500         0.34320
100        1.10060         0.85680         0.93160
1000       18.45930        1.38780         1.83710
10000      78.26120        1.75640         2.14200
100000     1545.29440      2.15530         2.60360

Prueba con objetivo que NO está en la lista:
Lineal: 412.007300008554 ms
Binaria Iterativa: 2.281900029629469 ms
Binaria Recursiva: 2.2326000034809113 ms

Resultado con lista pequeña:
Resultado búsqueda lineal: 1
Resultado binaria iterativa: 1
Resultado binaria recursiva: 1

Comparación de tiempos de ordenamiento (en segundos):
Tamaño     Burbuja         Selección       Sorted()
100        0.00040         0.00020         0.00001
500        0.01236         0.00546         0.00005
1000       0.05397         0.02114         0.00015
```

Aplicación en sistema de gestión de empleados

Se implementó un sistema en Python para gestionar empleados, donde los algoritmos analizados se aplican directamente. El sistema permite:

- Agregar empleados manualmente (guardados en un archivo CSV)
- Cargar empleados desde un archivo existente
- Mostrar todos los empleados
- Ordenarlos por salario (usando sorted())
- Buscar por apellido (búsqueda lineal)
- Buscar por rango de edad (filtro condicional)

empleados.py: contiene todas las funciones del sistema (lectura, escritura, búsqueda, ordenamiento)

```
# empleados.py
import csv

def ingresar_empleados():
```

```
    """
    Lee los empleados desde un archivo CSV y devuelve una lista de diccionarios.
    """
    empleados = []
    with open('empleados.csv', newline='', encoding='utf-8') as archivo:
        lector = csv.DictReader(archivo)
        for fila in lector:
            empleados.append({
                "nombre": fila["nombre"],
                "apellido": fila["apellido"],
                "edad": int(fila["edad"]),
                "salario": float(fila["salario"])
            })
    return empleados

def agregar_empleados():
    """
    Permite agregar nuevos empleados por teclado y los guarda en empleados.csv.
    """
    cantidad = int(input("¿Cuántos empleados nuevos deseas agregar? "))

    with open('empleados.csv', 'a', newline='', encoding='utf-8') as archivo:
        campos = ['nombre', 'apellido', 'edad', 'salario']
        escritor = csv.DictWriter(archivo, fieldnames=campos)

        for i in range(cantidad):
            print(f"\nIngrese datos del empleado {i + 1}:")
            nombre = input("Nombre: ")
            apellido = input("Apellido: ")
            edad = int(input("Edad: "))
            salario = float(input("Salario: "))
            escritor.writerow({
                'nombre': nombre,
                'apellido': apellido,
                'edad': edad,
                'salario': salario
```

```
        })

    print("\n Empleados agregados correctamente.")

def mostrar_empleados(empleados):
    """
    Muestra una lista de empleados en formato legible.
    """
    for emp in empleados:
        print(f"{emp['nombre']} {emp['apellido']} - Edad: {emp['edad']}, Salario: ${emp['salario']:,.2f}")

def ordenar_por_salario(empleados):
    return sorted(empleados, key=lambda emp: emp["salario"], reverse=True)

def buscar_empleados_por_edad(empleados):
    edad_min = int(input("Edad mínima: "))
    edad_max = int(input("Edad máxima: "))
    resultados = [emp for emp in empleados if edad_min <= emp["edad"] <= edad_max]

    if resultados:
        print(f"\nEmpleados entre {edad_min} y {edad_max} años:")
        mostrar_empleados(resultados)
    else:
        print("No se encontraron empleados en ese rango.")

def busqueda_lineal_por_apellido(empleados, apellido):
    for emp in empleados:
        if emp["apellido"].lower() == apellido.lower():
            return emp
    return None
```

main_empleados.py: interfaz de usuario en consola para acceder a las funciones

```
from empleados import (
    ingresar_empleados,
    agregar_empleados,
    mostrar_empleados,
    ordenar_por_salario,
    buscar_empleados_por_edad,
    busqueda_lineal_por_apellido
)

def main():
    print("GESTIÓN DE EMPLEADOS")

    respuesta = input("¿Querés agregar nuevos empleados? (s/n): ").lower()
    if respuesta == 's':
        agregar_empleados()

    empleados = ingresar_empleados()

    print("\n Lista actual de empleados:")
    mostrar_empleados(empleados)

    print("\n Empleados ordenados por salario (descendente):")
    empleados_ordenados = ordenar_por_salario(empleados)
    mostrar_empleados(empleados_ordenados)

    print("\n Buscar empleados por rango de edad:")
    buscar_empleados_por_edad(empleados)

    print("\n Buscar por apellido:")
    apellido = input("Ingrese el apellido a buscar: ")
    encontrado = busqueda_lineal_por_apellido(empleados, apellido)
    if encontrado:
        print(f"Empleado encontrado: {encontrado['nombre']} {encontrado['apellido']}, Salario:
${encontrado['salario']}")
    else:
```

```
        print("No se encontró un empleado con ese apellido.")

if __name__ == "__main__":
    main()
```

Decisiones de diseño

- Se modularizó el proyecto en archivos separados para mejorar su mantenibilidad.
- Se eligieron los algoritmos en base a pruebas de rendimiento reales.
- Se usó sorted() para ordenar empleados por salario o apellido, debido a su alta eficiencia.
- Se mantuvo búsqueda lineal por apellido por su simplicidad en listas pequeñas o no ordenadas.
- La lógica permite fácilmente escalar o reemplazar algoritmos según el crecimiento del sistema.

Validación

Se comprobó que todos los algoritmos respondieron correctamente ante valores presentes y ausentes.

Las búsquedas binaria iterativa y recursiva fueron significativamente más rápidas en listas grandes.

Los ordenamientos clásicos fueron útiles para comparar, pero sorted() resultó ser el más eficiente.

El sistema de empleados funcionó correctamente integrando todos estos componentes.

Metodología Utilizada

Investigación previa

El desarrollo del proyecto comenzó con una investigación teórica sobre:

Algoritmos de búsqueda (lineal y binaria).

Algoritmos de ordenamiento clásicos (burbuja, selección) y el algoritmo sorted() de Python.

Medición de rendimiento con timeit.

Aplicaciones prácticas de algoritmos en estructuras de datos.

Se consultaron fuentes como la documentación oficial de Python, videos educativos, y el material brindado por la cátedra.

Etapas del desarrollo

Diseño inicial

Se definió una estructura modular en archivos separados para facilitar el mantenimiento y la legibilidad. Se planificó realizar pruebas sobre diferentes tamaños de listas y casos prácticos.

Implementación

Se desarrollaron los siguientes módulos:

busquedas.py: búsqueda lineal, binaria iterativa y binaria recursiva.

ordenamientos.py: ordenamiento burbuja, selección y uso de sorted().

generador.py: generación de listas numéricas ordenadas aleatorias.

pruebas.py: funciones para medir tiempos con timeit.

main.py: pruebas comparativas entre algoritmos.

empleados.py y main_empleados.py: sistema de gestión de empleados.

Herramientas utilizadas

Lenguaje: Python 3

Librerías estándar: random, timeit, csv

IDE: Visual Studio Code

Control de versiones: Git y GitHub

Visualización: Google Sheets para gráficos y comparación de resultados

Documentación: Google Drive para colaboración en el informe PDF

Trabajo colaborativo

Nos dividimos las tareas: una integrante implementa las funciones de búsqueda binaria y la otra la búsqueda lineal.
Además, una se encargó de agregar los algoritmos de ordenamiento (burbuja, selección y sorted()), así como de desarrollar la base de la aplicación práctica, un sistema de gestión de empleados que permite cargar, ordenar y buscar empleados por diferentes criterios.

La otra integrante se ocupó de modularizar el proyecto, adaptar el código original al nuevo enfoque propuesto por la cátedra, e integrar la lectura y escritura de empleados mediante archivos CSV, mejorando la organización, persistencia de datos y reutilización del sistema.

Ambas revisamos el código y trabajamos juntas en la redacción del informe.
Para gestionar el código y la documentación de manera organizada, se utilizó un repositorio compartido, facilitando el control de versiones y el trabajo colaborativo.

Evaluación del rendimiento y aplicación práctica

Los resultados obtenidos muestran con claridad las diferencias de rendimiento entre los algoritmos de búsqueda implementados, y cómo esa información fue clave para diseñar el sistema de gestión de empleados.

Búsqueda binaria iterativa

- Fue el algoritmo con mejor rendimiento general, incluso en listas grandes.

- Su tiempo se mantuvo prácticamente constante (~2 ms) a partir de listas de 1.000 elementos.

- Estos resultados coinciden con su complejidad logarítmica (O(log n)), lo que lo hace ideal para estructuras ordenadas.

Búsqueda binaria recursiva

- También presentó una excelente performance, aunque ligeramente inferior por la sobrecarga de llamadas recursivas.

- Mantuvo tiempos bajos en todos los tamaños, siendo una alternativa válida cuando se prioriza legibilidad del código.

Búsqueda lineal

- En listas pequeñas fue rápida, pero su rendimiento decayó fuertemente en listas grandes.

- Por ejemplo, en una lista de 100.000 elementos tardó más de 1.500 ms, mientras que la binaria resolvió en apenas 2 ms.

- Esto confirma su complejidad lineal (O(n)), poco eficiente para búsquedas sobre volúmenes grandes.

Prueba con valor inexistente

Incluso cuando el objetivo no se encuentra en la lista, la búsqueda binaria mantuvo su eficiencia:

- Lineal: 412 ms

- Binaria iterativa: 2.28 ms

- Binaria recursiva: 2.23 ms

Esto refuerza que la búsqueda binaria es superior incluso en el peor caso.

Aplicación práctica en el sistema de empleados

Los resultados anteriores fueron clave para implementar un sistema funcional que aprovecha las ventajas de cada algoritmo:

Se utilizó sorted() para ordenar empleados por salario o apellido, debido a su eficiencia demostrada en las pruebas de ordenamiento.

Se aplicó búsqueda lineal por apellido en listas pequeñas, por su sencillez y porque no requería ordenar previamente.

Se implementó filtrado por rango de edad, sin necesidad de ordenar la lista, aprovechando la eficiencia de un recorrido secuencial acotado.

Estas decisiones fueron posibles gracias al análisis empírico, demostrando la importancia de evaluar algoritmos antes de aplicarlos en una solución real.

Errores corregidos

Inicialmente, la función recursiva no controlaba correctamente el caso base. Se soluciona agregando una condición para evitar errores.

Enlace al repositorio

El código fuente completo y los resultados se encuentran disponibles en el siguiente repositorio:

https://github.com/anahi651/TrabajoIntegradorProgramaci-n.git

Conclusiones

Este trabajo nos permitió comprender en profundidad el funcionamiento y la eficiencia de distintos algoritmos de búsqueda y ordenamiento, no solo desde un enfoque teórico, sino también a través de pruebas empíricas y su aplicación en un sistema concreto.

A partir de este análisis, pudimos seleccionar e implementar las estrategias más adecuadas para resolver un problema real, como la gestión de empleados, demostrando cómo el conocimiento algorítmico impacta directamente en el rendimiento y la calidad del software.

Durante el desarrollo, aprendimos a:

- Implementar, comparar y evaluar algoritmos clásicos de búsqueda y ordenamiento.

- Medir el rendimiento de forma precisa utilizando herramientas como timeit.

- Organizar el código de manera modular, clara y reutilizable, facilitando su mantenimiento y expansión.

- Aplicar conceptos teóricos en un entorno práctico, desarrollando una solución funcional basada en datos reales.

Este proceso integrador nos permitió no solo afianzar conocimientos técnicos, sino también desarrollar habilidades de análisis, diseño y colaboración que resultan esenciales en la formación profesional en programación.

Utilidad del tema

El análisis y comparación de algoritmos es fundamental en programación, ya que permite elegir soluciones más eficientes según el contexto del problema.

Estos conocimientos pueden aplicarse en cualquier proyecto donde sea necesario buscar elementos dentro de estructuras de datos.

Posibles mejoras y extensiones

Agregar algoritmos de ordenamiento para comparar no solo la eficiencia de búsqueda, sino el impacto del ordenamiento.

Combinar el análisis de búsqueda y ordenamiento, evaluando cómo influye la elección del algoritmo de ordenamiento en el rendimiento global cuando se trabaja con listas desordenadas.

Realizar visualizaciones gráficas del rendimiento de los algoritmos utilizando librerías.

Dificultades y resolución

Resolver errores en la implementación recursiva.

Ajustar las pruebas para obtener resultados más precisos.

Coordinar el trabajo en equipo usando herramientas como Git.

Posibles mejoras y extensiones

Validación de entradas del usuario: agregar controles para evitar que se ingresen datos inválidos como edades negativas, sueldos no numéricos o campos vacíos.

Sistema de menú interactivo: permitir que el usuario realice múltiples acciones (agregar, ordenar, buscar, salir) sin tener que reiniciar el programa después de cada operación.

Agregar más opciones de ordenamiento: permitir ordenar también por edad o apellido, según la necesidad del usuario.

Implementar búsqueda binaria por apellido, siempre que la lista esté ordenada alfabéticamente, lo que mejoraría la eficiencia en listas grandes.

Exportación de resultados: permitir guardar los empleados filtrados y ordenados en un nuevo archivo (por ejemplo, empleados_filtrados.csv).

Bibliografía

Búsqueda y ordenamiento en programación. (s.f.). [PDF]. Material de estudio proporcionado por la cátedra de Programación I.

Tecnicatura. (2025, 19 mayo.). Introdución Busqueda y Ordenamiento- Integrador YouTube. https://www.youtube.com/watch?v=u1QuRbx-_x4

Tecnicatura. (2025, 27 mayo.). Comparación de algoritmos de búsqueda en Python. YouTube. https://www.youtube.com/watch?v=haF4P8kF4Ik

Python Software Foundation. (2024). Python documentation (Versión 3.12). https://docs.python.org/