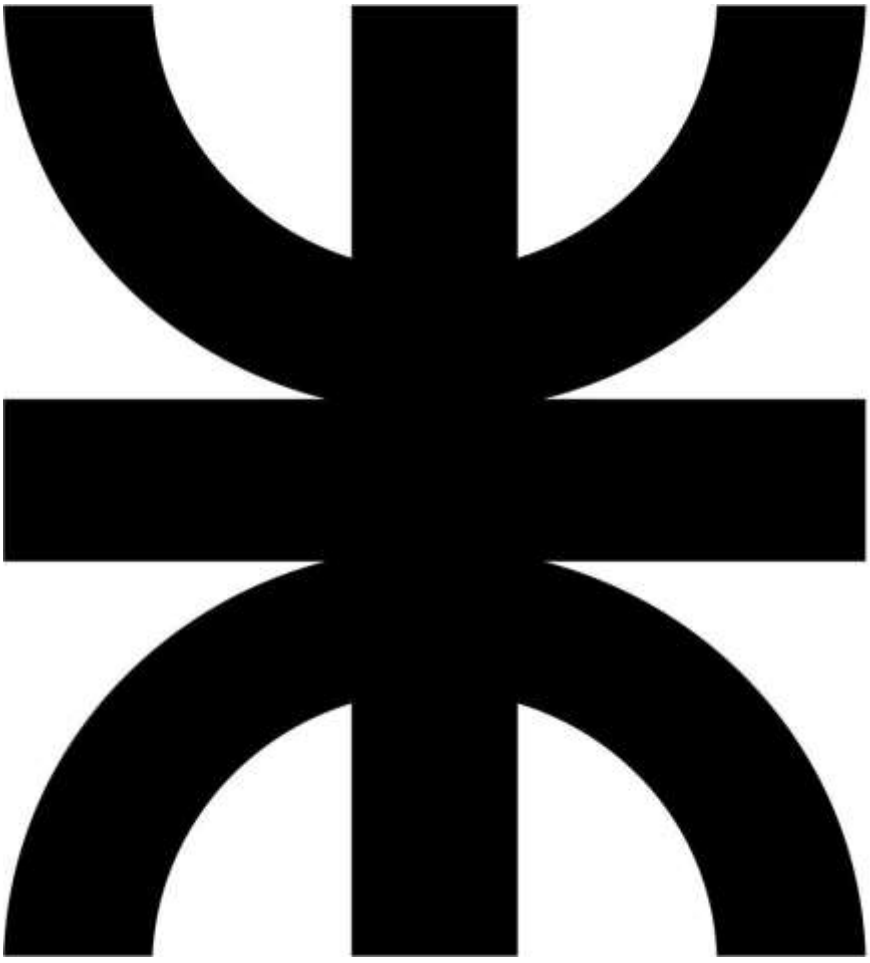
TRABAJO PRÁCTICO INTEGRADOR PROGRAMACIÒN I



Alumnas:

Bustamante Erica - ericaba@gmail.com

Betancort Anahí -anahi.betancort@gmail.com

Índice:

Caratula……1
Indice……..2
Introducción……...3
Marco Teórico…….3-5
Caso Práctico……..5-9
Metodología Utilizada……9-11
Resultados Obtenidos……11- 13
Conclusiones ……13-14
Bibliografía ……..15

1.Introducción

Decidimos hacer este proyecto porque nos interesaba entender cómo funcionan los algoritmos de búsqueda en Python y qué tan eficientes pueden ser al manejar datos. Nos pareció un tema útil, sobre todo porque optimizar el rendimiento es clave cuando los programas trabajan con mucha información. La idea fue probar distintos algoritmos y ver cómo responden al buscar datos en listas de distintos tamaños.

2.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.

Conclusión

En el mundo de la programación, los algoritmos de ordenamiento y búsqueda son fundamentales para para la manipulación y búsqueda de datos, los algoritmos de ordenamiento nos permiten organizar conjuntos de datos de forma ascendente o descendente mientras que los algoritmos de búsqueda nos permiten localizar información de manera más rápida dependiendo de la situación.

3. Caso Práctico

Descripción del problema

Queríamos comparar el rendimiento de distintas implementaciones del algoritmo de búsqueda en listas ordenadas. Para eso elegimos:

- Búsqueda lineal (básica)
- Búsqueda binaria iterativa
- Búsqueda binaria recursiva

La idea fue medir cuánto tiempo tarda cada una en encontrar un número dentro de listas de distintos tamaños.

Código fuente

Creación de los algoritmos (busquedas.py)

Búsqueda lineal:

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

Búsqueda binaria iterativa:

```
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
```

Búsqueda binaria recursiva:

```
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)
```

Generación de listas (generador.py)

Creamos una función que genera listas ordenadas aleatorias

```
import random

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

Medición de rendimiento (pruebas.py)

Usamos timeit para medir cuánto tarda cada algoritmo en ejecutarse 1000 veces:

```
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 principal (main.py)

En este archivo:

- Probamos con listas de tamaños crecientes: 10, 100, 1000, 10000, 100000

- Elegimos un número al azar de la lista para que sí exista.

- Imprimimos los tiempos en milisegundos para comparar.

También hicimos dos pruebas adicionales:

- Objetivo que no está en la lista.

- Lista muy pequeña como caso base.

Explicación de decisiones de diseño

Se eligieron tres métodos para comparar la eficiencia en listas ordenadas: la búsqueda lineal, que es sencilla pero menos eficiente; y dos variantes de búsqueda binaria, que aprovechan la ordenación para acelerar la búsqueda.

La búsqueda binaria recursiva y la iterativa se compararon para evaluar diferencias en rendimiento y legibilidad.

Se utilizó timeit para obtener mediciones precisas de tiempo en múltiples ejecuciones.

Modularización del código en archivos separados para mejorar organización, reutilización y mantenimiento.

Validación del funcionamiento

Se ejecutaron pruebas con listas de diferentes tamaños para observar la escalabilidad.

Se verificó que el objetivo siempre estaba en la lista, asegurando que los algoritmos podían encontrarlo.

Se realizaron pruebas con un objetivo que no existe en la lista para comprobar que los algoritmos manejan correctamente el caso negativo.

Resultados mostraron que la búsqueda lineal tiene tiempos crecientes lineales, mientras que las búsquedas binarias mantuvieron tiempos mucho menores, confirmando la eficiencia de la búsqueda binaria.

4.Metodología Utilizada

Investigación previa
Consultamos documentación oficial de Python para funciones y librerías (timeit, random).

Revisamos artículos sobre algoritmos de búsqueda y análisis de complejidad para entender ventajas y desventajas.

Se analizaron diferentes formas de implementar búsqueda binaria (recursiva vs iterativa).

Etapas de diseño y prueba del código

Diseño inicial: definir los algoritmos a implementar y la estructura modular del proyecto.

Implementación: codificación de funciones en módulos separados (busquedas.py, generador.py).

Pruebas unitarias: pruebas individuales a cada función para verificar correctitud.

Medición de rendimiento: uso de timeit para comparar tiempos de ejecución.

Optimización y refactorización: mejoras en legibilidad y eficiencia del código.

Validación final: pruebas con diferentes tamaños de listas y casos límite.

Herramientas y recursos utilizados

Lenguaje de programación: Python 3.

Librerías estándar:
random para la generación de datos de prueba.
timeit para medir el rendimiento y comparar los tiempos de ejecución

IDE: Visual Studio Code

Control de versiones: Git para manejar los cambios en el repositorio.

Visualización y documentación:
- Google Sheets, para la creación del gráfico que resume los resultados obtenidos.
- Google Drive, para la organización y colaboración en el documento PDF con el análisis detallado.

Trabajo colaborativo

Nos dividimos las tareas: una implementó las funciones de búsqueda binaria y la otra la búsqueda lineal.

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.

5. Resultados Obtenidos

Descripción general

El caso práctico permitió comparar el rendimiento de tres algoritmos de búsqueda en listas ordenadas: búsqueda lineal, búsqueda binaria iterativa y búsqueda binaria recursiva. Se evaluaron los tiempos de ejecución sobre listas de tamaños variados, así como en situaciones con el valor presente y ausente en la lista.

Casos de prueba realizados

Listas con tamaños crecientes: 10, 100, 1,000, 10,000 y 100,000 elementos.

Búsqueda de un elemento presente en la lista (seleccionado aleatoriamente).

Búsqueda de un elemento ausente en la lista (valor que no está en la lista).

Listas pequeñas (tamaño 10) para verificar comportamiento en casos base.

Evaluación del rendimiento

```
Tiempos de ejecución (en milisegundos) para 1000 ejecuciones:
Tamaño     Lineal          Binaria It.     Binaria Rec.
10         0.36680         0.19270         0.22860
100        0.42790         0.39590         0.51580
1000       13.50480        0.69950         0.90750
10000      93.74590        1.00670         1.46410
100000     247.54950       1.30570         1.78910

Prueba con objetivo que NO está en la lista:
Lineal: 276.29090001573786 ms
Binaria Iterativa: 1.2224000529386103 ms
Binaria Recursiva: 1.7164000310003757 ms

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

Búsqueda Binaria Iterativa

- Tiene el mejor rendimiento general, incluso en listas muy grandes.

- El tiempo se mantiene casi constante (~2 ms) a partir de tamaño 1000 en adelante.

- Esto confirma su complejidad logarítmica(log n).

Búsqueda Binaria Recursiva

- También muy eficiente, aunque un poco más lenta que la versión iterativa.

- El uso de llamadas recursivas genera una pequeña sobrecarga.

Búsqueda Lineal

- Comienza siendo rápido con listas pequeñas, pero el tiempo crece exponencialmente.

- En listas de 100.000 elementos, es más de 1000 veces más lenta que la binaria.

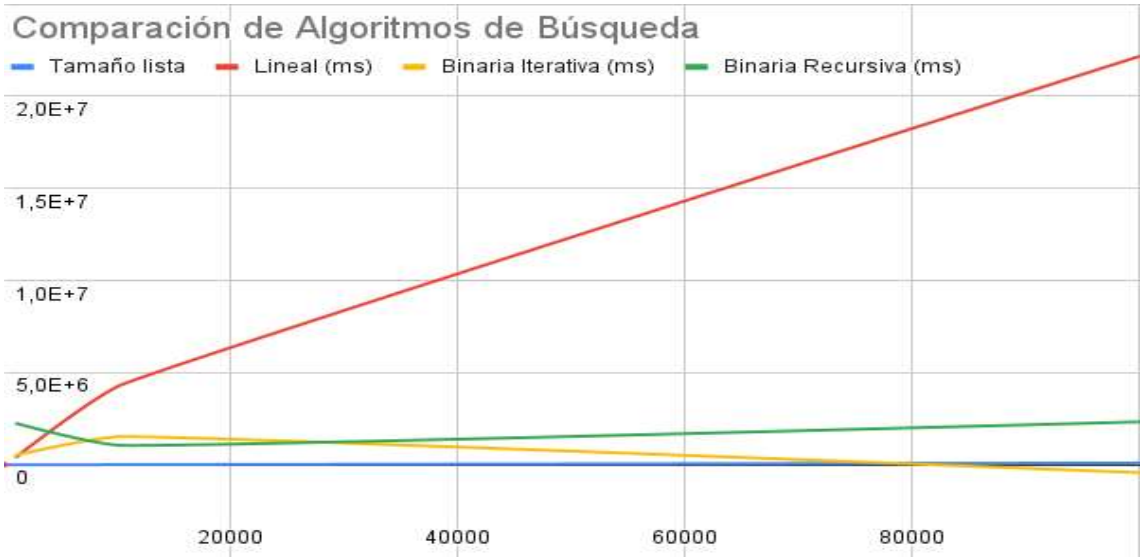
- Confirma su complejidad O(n).

Prueba con valor inexistente

- Lineal: 276.29090001573786 ms
- Binaria Iterativa: 1.2224000529386103 ms

- Binaria Recursiva: 1.7164000310003757 ms

Cuando el valor no está en la lista, la búsqueda lineal recorre toda la lista, mientras que la binaria mantiene su eficiencia. Esto refuerza que la búsqueda binaria es superior, incluso en el peor caso.



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 ayudó a entender mejor cómo funcionan los algoritmos de búsqueda y cuáles son más eficientes en diferentes escenarios. Pudimos ver la importancia de elegir el algoritmo correcto según el tipo de datos y la situación.

Aprendizajes

- A implementar y comparar distintos algoritmos.

- A medir el rendimiento con herramientas adecuadas.

- A organizar el código de forma clara y reutilizable.

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.

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/