TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Algoritmos de Búsqueda
y
Ordenamiento en Python

Integrantes del grupo:

- **Joaquin Lencina**
- **Gabriela Mancini**

Profesor/a: *Cinthia Rigoni*

Materia: *Programación I*

Fecha de Entrega: *20/06/2025*

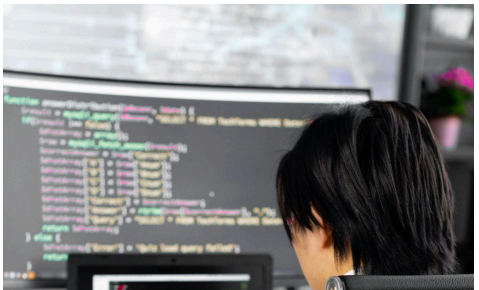
Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7 . Anexos

1. Introducción

Este trabajo integrador desarrolla la implementación de algoritmos de búsqueda y ordenamiento en el lenguaje de programación Python. Este tema no solo es de suma importancia en el ámbito laboral y académico, sino también es necesario conocer para desenvolverse en procesos de selección y entrevistas técnicas en el campo laboral del software.

Estos algoritmos son un componente habitual en las entrevistas técnicas de empresas tecnológicas. Saber



implementar y comparar estos algoritmos no solo demuestra conocimientos técnicos sólidos, sino también habilidades de análisis, lógica y optimización.

Muchas veces se recurre a plataformas para practicar las habilidades de programación, prepararse para entrevistas técnicas y participar en competencias de código. A la vez, muchas empresas las utilizan para evaluar candidatos con pruebas técnicas.

Entre estas plataformas tenemos: HackerRank, CoderByte, Codewars, LeetCode

Con este trabajo, se busca:

- Comprender el funcionamiento de los algoritmos de búsqueda y ordenamiento más utilizados.
- Aplicarlos en un caso práctico programado en Python.
- Analizar su rendimiento y adecuación a diferentes contextos.
- Valorar su relevancia como base en desafíos técnicos reales.

2. Marco Teórico

Este apartado contiene la fundamentación conceptual del tema tratado. Debe incluir definiciones, clasificaciones, jerarquías, estructuras y sintaxis si corresponde.

Notación Big O

Antes de entrar en estos conceptos entendamos uno muy importante es es la complejidad de cada uno en cuanto al rendimiento de los algoritmos. Para esos usamos la notación Big O que nos indica cuántas operaciones necesita hacer un algoritmo para llegar a la resolución y cómo crece ese número de operaciones cuando la cantidad de datos aumenta

Algoritmos de Búsqueda

Un algoritmo de búsqueda es un paso a paso, una serie de instrucciones, para encontrar algo dentro de un conjunto de datos, como por ejemplo una lista, un archivo, un diccionario o una base de datos.

- **Búsqueda Lineal**: Recorre la lista elemento por elemento hasta encontrar el valor buscado. Vas leyendo uno por uno desde el principio hasta que encontrás lo que buscás (o llegás al final). Es como buscar el nombre de una persona en una lista desordenada. Tiene complejidad O(n) siendo n el número de

pasos, por lo que significa que si duplicás la cantidad de datos, el algoritmo tarda más o menos el doble de tiempo.

- **Búsqueda Binaria**: solamente funciona si la lista está ordenada.En lugar de mirar uno por uno, dividís la lista a la mitad y te fijás si lo que buscás está antes o después.
  Es como buscar una palabra en el diccionario: sabés que está ordenado, así que no empezás por la A si buscás la palabra “vino”. Complejidad O(log n).

Algoritmos de Ordenamiento

Un algoritmo de ordenamiento es un conjunto de pasos que sirve para poner en orden una lista de cosas, como números, palabras, nombres o fechas. Existen varias opciones para ordenar y varían en rapidez, complejidad y cantidad de datos que manejan. Elegimos uno u otro dependiendo de cuántos datos tenemos y qué tan ordenados están desde el inicio.

- **Burbuja (Bubble Sort)**: tiene ese nombre porque al comparar los números de una lista, por ejemplo, los números más grandes van subiendo al final de la lista, como si

fueran burbujas que flotan hacia arriba. El proceso es ir comparando los elementos de a pares y cambiándolos de lugar si están en el orden incorrecto. Es ineficiente para listas grandes ($O(n^2)$).

[4, 2, 7, 1] => [2, 4, 7, 1] => [2, 4, 1, 7] => [2, 1, 4, 7] => [1, 2, 4, 7]

- **Selección (Selection Sort)**: es un método para ordenar una lista buscando el número más chico y poniéndolo en su lugar. Después vuelve a buscar el siguiente más chico y también lo acomoda, y así sucesivamente hasta que la lista queda ordenada. Siempre busca el mínimo en la parte que falta ordenar.También $O(n^2)$.

  [4, 2, 7, 1] => [1, 4, 2, 7] => [1, 2, 4, 7]

- **Inserción (Insertion Sort)**: ordena una lista de números comparando cada número con los que están antes y colocándolo en el lugar correcto. Es como ordenar un mazo de cartas, tomando la segunda y la comparás con la que sigue y la colocó a la izquierda o derecha según estés ordenando de menora a mayor. Seguís así consecutivamente hasta llegar al final. $O(n^2)$ en el peor caso, eficiente con listas pequeñas o casi ordenadas.

  [4, 2, 7, 1] => [2, 4, 7, 1] => [2, 4, 7, 1] = > [2, 4, 1, 7] => [2, 1, 4, 7] => [1, 2, 4, 7]

- **Merge Sort:** es un algoritmo para ordenar listas que funciona con la idea de dividir la lista en partes pequeñas, ordenar esas partes (que ya son fáciles de manejar y unir (merge) las partes ordenadas en una lista grande y ordenada. Complejidad O(n log n), es decir no recorre toda la lista muchas veces, pero sí hace más que una sola pasada.

  [4, 2, 7, 1] =>[4, 2] y [7, 1] => [4], [2], [7], [1] => [2, 4] y [1, 7] => [1, 2, 4, 7]

- **Quicksort**: Funciona dividiendo la lista en partes más chicas usando un elemento especial llamado pivote, y acomodando los demás elementos alrededor de ese pivote. Ese pivote es un número cualquiera de la lista (por ejemplo, el primero). Separás la lista en dos grupos: menores que el pivote - mayores que el pivote. Seguimos ordenando cada grupo repitiendo el mismo proceso. Cuando está ordenado juntamos todo menores + pivote + mayoresRápido en la práctica, con O(n log n) promedio y O(n^2) en el peor caso.

  [4, 2, 7, 1] => pivote [4] => [1, 2] [4] [7] => [1, 2, 4, 7]

3. Caso Práctico

A continuación se presentarán casos prácticos en desafíos que podemos encontrar en entrevistas técnicas cuando aspiramos a un puesto de IT:

Búsqueda

Desafío:: vamos a buscar en una lista de personas un nombre específico

```
personas = [
    {"nombre": "Ana", "edad": 25},
    {"nombre": "Luis", "edad": 32},
    {"nombre": "Pedro", "edad": 29},
    {"nombre": "María", "edad": 40},
    {"nombre": "Carlos", "edad": 19}
]
```

Con búsqueda lineal

Vas uno por uno, como cuando leés nombres en una lista de invitados.

```
def buscar_por_nombre_lineal(lista, nombre_buscado):
    for persona in lista:
        if persona["nombre"] == nombre_buscado:
            return persona
    return None
resultado = buscar_por_nombre_lineal(personas, "María")
print(resultado)
```

Con búsqueda binaria

Divide la lista por la mitad y compara

```
def buscar_binaria_por_nombre(lista, nombre_buscado):
    izquierda = 0
    derecha = len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        actual = lista[medio]["nombre"]
        if actual == nombre_buscado:
            return lista[medio]
        elif actual < nombre_buscado:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return None
```

```
resultado = buscar_binaria_por_nombre(personas_ordenadas,
"María")
print(resultado)
```

En ambos el resultado es `{'nombre': 'María', 'edad': 40}`

La diferencia es que en la lineal funciona con listas desordenadas y puede tardar más si la lista es muy larga. En el caso de la binaria es mucho más rápida si la lista está ordenada, pero no funciona si la lista no está ordenada.

```
        return not set(lista1).isdisjoint(lista2)
```

Ordenamiento

Desafío: verificar si dos palabras son anagramas, pero sin usar sorted().

Función base:

```
def son_anagramas(str1, str2):
    if len(str1) != len(str2):
        return False
    return ordenar(str1) == ordenar(str2)
```

Bubble Sort

```
    def bubble_sort(s):
        lista = list(s)
        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
```

Selection Sort

```
    CopiarEditar
    def selection_sort(s):
        lista = list(s)
        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
```

Insertion Sort

```
def insertion_sort(s):
    lista = list(s)
    for i in range(1, len(lista)):
        actual = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > actual:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual
    return lista
```

Merge Sort

```
def merge_sort(s):
    lista = list(s)
    if len(lista) <= 1:
        return lista

    def merge(left, right):
        resultado = []
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                resultado.append(left[i])
                i += 1
            else:
                resultado.append(right[j])
                j += 1
        resultado.extend(left[i:])
        resultado.extend(right[j:])
        return resultado

    medio = len(lista) // 2
    izquierda = merge_sort(lista[:medio])
    derecha = merge_sort(lista[medio:])
    return merge(izquierda, derecha)
```

Quicksort

```
def quicksort(s):
    lista = list(s)
    if len(lista) <= 1:
        return lista
    pivote = lista[0]
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)
```

4. Metodología Utilizada

El desarrollo de este trabajo práctico se llevó a cabo siguiendo una metodología estructurada en distintas etapas, que incluyó exploración previa del tema, planificación pedagógica del encuadre, codificación y confección de materias audiovisuales.

El primer paso fue una investigación teórica sobre los distintos tipos de algoritmos de búsqueda y ordenamiento más comunes en programación. Materiales de clase y artículos en internet.

Cada algoritmo fue escrito sin utilizar funciones de ordenamiento integradas (sorted() o .sort()), para reforzar la comprensión lógica del funcionamiento interno.

Para cada tipp de algoritmo (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort y Quicksort), se diseñaron funciones separadas para una mejor comprensión de su funcionamiento utilizando siempre la misma lista de ejemplo.

Por último, se desarrollaron funciones para aplicar estos algoritmos en ell desafío de verificar si dos palabras son anagramas. Se resuelve esta tarea aplicando los diferentes ordenamientos

Herramientas y recursos utilizados

- Lenguaje: Python 3.12
- Entorno de desarrollo: Visual Studio Code
  Control de versiones: Git (uso local)
- Capturas y documentación: Google Docs y Markdown para redacción del trabajo

5. Resultados Obtenidos

A lo largo del desarrollo del trabajo práctico, se lograron implementar y validar con éxito los distintos algoritmos de búsqueda y ordenamiento seleccionados: búsqueda lineal, búsqueda binaria, bubble sort, selection sort, insertion sort, merge sort y quicksort.

Los algoritmos fueron probados en casos concretos, como el desafío de verificar si dos palabras son anagramas y el desafío de buscar un objeto con los datos de una persona utilizando tanto una lista desordenada (para búsqueda lineal) como una lista ordenada (para

búsqueda binaria). Esto permitió comprobar en la práctica cómo el tipo de algoritmo y el estado de los datos afectan el resultado y el rendimiento.

6. Conclusiones

La realización de este trabajo práctico permitió mejorar una comprensión profunda y aplicada de los algoritmos de búsqueda y ordenamiento, tanto desde el punto de vista teórico como práctico.

¿Por qué es importante conocer estos conceptos?

- Escribir programas más eficientes.
- Elegir el método correcto para cada situación (no todo se resuelve igual).
- Evitar soluciones lentas o que se rompen con muchos datos.

8. Anexos

• Enlace al video explicativo.
• Repositorio.
• Presentación diapositivas