Trabajos Integradores – Programación I

Datos Generales

- **Título del trabajo:** Algoritmo de Búsqueda y Ordenamiento
- **Alumnos:** Sebastián Nicolás, Gossos Scribe y Marcelo, Gomez Armoa
- **Materia:** Programación I
- **Profesor/a:** AUS Bruselario, Sebastián
- **Fecha de Entrega:** 09/06/2025

UTN
UNIVERSIDAD TECNOLÓGICA NACIONAL

Índice

1. Introducción

El presente trabajo aborda la temática de los algoritmos de búsqueda y ordenamiento, fundamentales en el área de la programación y el manejo eficiente de datos.

Se eligió este tema debido a su importancia transversal en el desarrollo de software, ya que la correcta gestión y manipulación de grandes volúmenes de información es un desafío constante en la informática actual.

El objetivo principal de este trabajo es comprender, implementar y comparar distintos algoritmos de búsqueda y ordenamiento, evaluando su rendimiento y aplicabilidad en diferentes contextos. Se busca que el alumno adquiera herramientas prácticas para seleccionar el algoritmo más adecuado según el problema a resolver, optimizando así el desempeño de sus aplicaciones.

2. Marco Teórico

Algoritmos de Búsqueda

Un algoritmo de búsqueda es un conjunto de instrucciones que permite localizar un elemento específico dentro de una colección de datos, como listas, arreglos o bases de datos, su eficiencia depende del tamaño de la colección y de si los datos están ordenados o no.

Los más comunes son:

Búsqueda lineal: consiste en recorrer secuencialmente todos los elementos de una colección hasta encontrar el valor deseado o en su defecto agotar la lista. Es simple y no requiere que los datos estén ordenados, pero su eficiencia disminuye en listas grandes.

```python
#! ---- ALGORITMOS DE BÚSQUEDA ---- #!

# * Búsca elemento por elemento en una lista desordenada (búsqueda lineal)
# Ineficiente para listas grandes, pero simple de implementar.
def busqueda_lineal(lista, elemento):
    for i in range(len(lista)):
        if lista[i] == elemento:
            return i
    return -1
```

Búsqueda binaria: Es mucho más eficiente, pero requiere una lista ordenada. El algoritmo compara el elemento central con el valor buscado y, según el resultado, descarta la mitad de la lista en cada paso.

Funcionamiento:

1- Compara el elemento central con el valor buscado.

2- Si son iguales, retorna la posición.

3- si el valor buscado es menor, repetir en la mitad izquierda.

4- si es mayor, repetir en la mitad derecha.

```python
# * Búsca un elemento de la lista partiendola en dos y luego compara el elemento con el del medio.
# Requiere que la lista esté ordenada, pero es mucho más eficiente que la búsqueda lineal.
def busqueda_binaria(lista_ordenada, elemento):
    izquierda = 0
    derecha = len(lista_ordenada) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista_ordenada[medio] == elemento:
            return medio
        elif lista_ordenada[medio] < elemento:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

Búsqueda por interpolación: es una mejora de la búsqueda binaria para listas ordenadas y distribuidas uniformemente. En vez de buscar siempre en el centro, estima la posición probable del elemento buscado usando una fórmula basada en la distribución de los valores.

Funcionamiento:

- Calcula la posición estimada del elemento según la proporción entre el valor buscado y los extremos de la lista.
- Si el valor en esa posición es el buscado, retorna la posición.
- Si es menor, repite en la parte derecha; si es mayor, en la izquierda.

```python
# * Similar a la búsqueda binaria pero más eficiente en listas ordenadas uniformemente distribuidas.
# Requiere que la lista esté ordenada y es más eficiente que la búsqueda binaria en ciertos casos.

def busqueda_interpolacion(lista_ordenada, elemento):
    izquierda = 0
    derecha = len(lista_ordenada) - 1
    while izquierda <= derecha and elemento >= lista_ordenada[izquierda] and elemento <= lista_ordenada[derecha]:
        if izquierda == derecha:
            if lista_ordenada[izquierda] == elemento:
                return izquierda
            return -1

        # Calcular posición estimada
        rango_indices = derecha - izquierda
        rango_valores = lista_ordenada[derecha] - lista_ordenada[izquierda]
        distancia_elemento = elemento - lista_ordenada[izquierda]

        # Proporción: ¿qué porcentaje del rango representa el elemento?
        proporcion = float(distancia_elemento) / rango_valores
        pos = izquierda + int(rango_indices * proporcion)

        if lista_ordenada[pos] == elemento:
            return pos
        if lista_ordenada[pos] < elemento:
            izquierda = pos + 1
        else:
            derecha = pos - 1
    return -1
```

Búsqueda exponencial: es útil cuando el tamaño de la lista es muy grande o desconocido. Primero encuentra un rango donde podría estar el elemento, duplicando el índice en cada paso (1, 2, 4, 8, ...), y luego realiza una búsqueda binaria en ese rango.

Búsqueda por saltos (Jump Search): La búsqueda por saltos es eficiente para listas ordenadas.

Divide la lista en bloques de tamaño fijo, salta de bloque en bloque hasta encontrar el rango donde podría estar el elemento, y luego realiza una búsqueda lineal dentro de ese bloque.

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son procedimientos que permiten reorganizar una colección de elementos según un criterio específico, como puede ser de menor a mayor, mayor a menor, o alfabéticamente. El ordenamiento es fundamental para optimizar búsquedas, facilitar el análisis de datos y mejorar la presentación de la información.

Tipos principales de algoritmos de ordenamiento

Bubble Sort (Ordenamiento burbuja):
Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista está completamente ordenada. Es sencillo de implementar, pero poco eficiente para listas grandes.

Insertion Sort (Ordenamiento por inserción):
Construye una lista ordenada tomando los elementos uno a uno e insertándolos en la posición adecuada dentro de la parte ya ordenada. Es eficiente para listas pequeñas o casi ordenadas.

Selection Sort (Ordenamiento por selección):
Selecciona el elemento más pequeño del arreglo y lo coloca en su posición final, repitiendo este proceso para el resto de la lista. Es fácil de entender, pero no es eficiente para grandes volúmenes de datos.

```python
# * El algoritmo de seleccion es un algoritmo de ordenamiento simple que divide la lista en dos partes
def ordenamiento_seleccion(lista):
    for i in range(len(lista)):
        min_index = i
        for j in range(i + 1, len(lista)):
            if lista[j] < lista[min_index]:
                min_index = j
        lista[i], lista[min_index] = lista[min_index], lista[i]
    return lista
```

Merge Sort (Ordenamiento por mezcla):

Utiliza la técnica de “divide y vencerás”. Divide la lista en sub-listas más pequeñas, las ordena y luego las combina para formar la lista ordenada. Es eficiente para listas grandes.

Quick Sort (Ordenamiento rápido):

También utiliza “divide y vencerás”. Selecciona un pivote y divide la lista en dos sub-listas: una con elementos menores y otra con elementos mayores al pivote. Luego ordena recursivamente ambas sub-listas.

```python
# * El algoritmo de quicksort es un algoritmo de ordenamiento eficiente que utiliza el enfoque de divide y vencerás.
def ordenamiento_quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[len(lista) // 2]
        izquierda = [x for x in lista if x < pivote]
        medio = [x for x in lista if x == pivote]
        derecha = [x for x in lista if x > pivote]
        return ordenamiento_quicksort(izquierda) + medio + ordenamiento_quicksort(derecha)
```

Aplicaciones

El ordenamiento es esencial para búsquedas eficientes (como la binaria), para análisis estadísticos (cálculo de medianas, percentiles), y para la presentación clara de datos en informes y visualizaciones.

En la práctica, los lenguajes de programación modernos incluyen funciones de ordenamiento optimizadas, pero conocer los algoritmos permite elegir la mejor estrategia según el contexto y los recursos disponibles.

3. Caso Práctico

El problema planteado consiste en simular la gestión y búsqueda eficiente de una gran cantidad de DNIs (20.000 números aleatorios) utilizando distintos algoritmos de ordenamiento y búsqueda. El objetivo es comparar el rendimiento y la eficacia de cada método, permitiendo al usuario seleccionar el algoritmo a utilizar y observar los resultados y tiempos de ejecución.

Diseño:

Se eligieron algoritmos clásicos (selección y quicksort para ordenamiento; lineal, binaria e interpolación para búsqueda) para comparar eficiencia y facilidad de implementación.

El menú interactivo permite al usuario experimentar con diferentes métodos y observar los resultados y tiempos de ejecución.

Se utiliza una copia de la lista original para cada prueba, asegurando independencia entre ejecuciones.

Se muestra el tiempo de ejecución para visualizar el impacto del algoritmo elegido.

Validación del funcionamiento:

Se realizaron pruebas con distintos valores y algoritmos, verificando que los resultados sean correctos y que los tiempos de ejecución sean coherentes con la teoría.

Se comprobó que los algoritmos de búsqueda requieren la lista ordenada cuando corresponde (binaria e interpolación).

El programa responde correctamente a entradas válidas e inválidas, mostrando mensajes claros al usuario.

4. Metodología Utilizada

Investigación previa: Se consultaron libros de algoritmos (Cormen et al.) y documentación oficial de Python.

Diseño y prueba: Se implementaron los algoritmos en Python y se probaron con listas generadas aleatoriamente.

Herramientas: Visual Studio Code como IDE, Python 3.12, y librerías estándar.

Trabajo colaborativo: El desarrollo, pruebas y documentación fueron realizados por los alumnos.

5. Resultados Obtenidos

Casos de prueba: Se ejecutaron los algoritmos con listas de 20.000 elementos y diferentes valores de búsqueda.

Errores corregidos: Se ajustaron detalles en la generación de listas y manejo de índices.

Evaluación de rendimiento:

Ordenamiento por selección: tiempo elevado en listas grandes.

Quicksort: tiempo significativamente menor.

Búsqueda binaria e interpolación: mucho más rápidas que la lineal en listas ordenadas.
Observaciones: El menú interactivo facilita la comparación y el aprendizaje práctico de los algoritmos.

Repositorio: https://github.com/SebaGossos/TP-Integrador-Programacion

6. Conclusiones

El trabajo permitió experimentar y comparar diferentes algoritmos de ordenamiento y búsqueda, observando en la práctica la importancia de elegir el método adecuado según el contexto y el tamaño de los datos.

Se comprobó que algoritmos eficientes como quicksort y búsqueda binaria mejoran notablemente el rendimiento en grandes volúmenes de datos.

Como mejora futura, se podrían implementar otros algoritmos y analizar su rendimiento en diferentes escenarios, así como agregar una interfaz gráfica para facilitar aún más la interacción.

Las dificultades surgidas, como la gestión de listas grandes y la validación de entradas, se resolvieron mediante pruebas y ajustes en el código.

7. Bibliografía

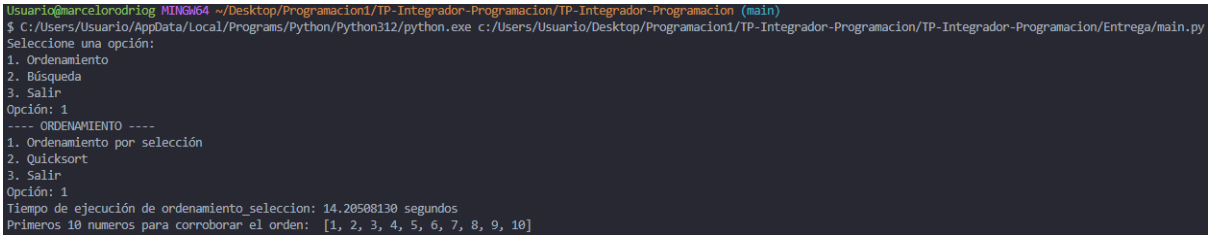
- Python Software Foundation. (2024). Python 3 Documentation. https://docs.python.org/3/
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- Sweigart, A. (2019). Automate the Boring Stuff with Python. No Starch Press.
- GeeksforGeeks. (2025). Searching Algorithms. https://www.geeksforgeeks.org/searching-algorithms/ (Accedido el 9 de junio de 2025)
- Apuntes de la cátedra de Programación I, UTN (2025).

8. Anexos

Material adicional que no va en el cuerpo principal del trabajo pero que aporta valor.

Capturas de pantalla del programa funcionando:

opc 1 = ordenamiento/ opc 1 = ordenamiento por selección.



opc 1 = ordenamiento/ opc 2 = quicksort

opc 2 = búsqueda / opc 1 lineal

opc 2 = búsqueda / opc 2 binaria

```
Usuario@marcelorodriog MINGW64 ~/Desktop/Programacion1/TP-Integrador-Programacion/TP-Integrador-Programacion (main)
$ C:/Users/Usuario/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Usuario/Desktop/Programacion1/TP-Integrador-Programacion/TP-Integrador-Programacion/Entrega/main.py
Seleccione una opción:
1. Ordenamiento
2. Búsqueda
3. Salir
Opción: 2
Ingrese el numero a buscar entre 1 y 20.000: 10000
---- INGRESE EL ALGORITMO DE BÚSQUEDA QUE PREFIERA ----
1. Búsqueda lineal
2. Búsqueda binaria
3. Búsqueda por interpolación
4. Salir
Opción: 2
Tiempo de ejecución de busqueda_binaria: 0.00000630 segundos
Resultado de la búsqueda binaria: 9999
```

opc 2 = búsqueda /opc 3 interpolación

```
Usuario@marcelorodriog MINGW64 ~/Desktop/Programacion1/TP-Integrador-Programacion/TP-Integrador-Programacion (main)
$ C:/Users/Usuario/AppData/Local/Programs/Python/Python312/python.exe c:/Users/Usuario/Desktop/Programacion1/TP-Integrador-Programacion/TP-Integrador-Programacion/Entrega/main.py
Seleccione una opción:
1. Ordenamiento
2. Búsqueda
3. Salir
Opción: 2
Ingrese el numero a buscar entre 1 y 20.000: 10000
---- INGRESE EL ALGORITMO DE BÚSQUEDA QUE PREFIERA ----
1. Búsqueda lineal
2. Búsqueda binaria
3. Búsqueda por interpolación
4. Salir
Opción: 3
Tiempo de ejecución de busqueda_interpolacion: 0.00001000 segundos
Resultado de la búsqueda por interpolación: 9999
```