TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

# Trabajo Práctico – Algoritmos de búsqueda y ordenamiento.

Presentado por:

Israel Garcia Moscoso – isragadiel@gmail.com
Juan Esteban Gelos – juan_gelos@yahool.com

Materia: Programación I
Profesor: Ariel Enferrel
Tutor: Ramiro Hualpa
Fecha de Entrega: 9 de junio de 2025

---

Índice



## 1. Introducción

Existen diferentes algoritmos de búsqueda, cada uno con sus propias ventajas y desventajas, a saber: búsqueda lineal; búsqueda binaria: búsqueda por interpolación; búsqueda de hash; Búsqueda por salto; búsqueda ternaria; búsqueda exponencial; búsqueda de ancho primero (BFS); búsqueda de profundidad primero (DFS); búsqueda Beam; búsqueda bidireccional; búsqueda A*; Búsqueda IDA∗ (Iterative Deepening A∗ Search); y búsqueda detección de ciclos.

Estos algoritmos se utilizan para la búsqueda de palabras clave en un documento, de archivos en un sistema de archivos, de registros en una base de datos, la ruta más corta en un gráfico o en soluciones a problemas de optimización.

Los algoritmos de ordenamiento, organizan los datos de acuerdo a un criterio definido, como de menor a mayor o alfabéticamente. Son importantes porque permiten organizar y estructurar datos de manera eficiente, permitiendo realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Entre los algoritmos de ordenamiento podemos citar: ordenamiento por burbuja; ordenamiento por selección; ordenamiento por inserción; ordenamiento rápido (Quicksort); ordenamiento por mezcla (Mergesort); Heapsort; shellsort; counting sort; radix sort; bucket sort; otros no basados en comparaciones como por ejemplo: Pigeonhole Sort; Gnome Sort; Comb Sort; Odd-Even Sort; Stooge Sort; Bogosort (Permutation Sort, Stupid Sort); Sleep Sort; Pancake Sort; y otros especializados, híbridos como: Timsort; e Introsort.

La elección del algoritmo de ordenamiento adecuado depende de varios factores, como el tamaño de la lista, el tipo de datos y los requisitos de rendimiento.

En este trabajo se realizó un análisis de los algoritmos de búsqueda lineal y binaria, y dentro de los de ordenamiento, los métodos  por burbuja; por selección; por inserción; y rápido o Quicksort, realizándose una comparación de tiempos de ejecución.

## 2. Marco Teórico

### Ordenamiento.

Existen múltiples algoritmos capaces de ordenar un arreglo u otro tipo de estructuras de datos. Cada uno de estos algoritmos se ajustan mejor como solución en determinados escenarios.

### 1. Ordenamiento por burbuja (bubble sort).

Es un algoritmo simple y fácil de instrumentar. Se basa en recorrer el vector de datos comparando los elemento adyacentes entre sí, intercambiandolos si el elemento de la derecha es mayor o menor según el criterio que se desee emplear. Esto debe repetirse una y otra vez sobre el vector hasta dejarlo ordenado.

La ventaja que presenta este método es su simple entendimiento e implementación, pero tiene la desventajas de ser muy ineficiente para listas grandes.

Mostramos aquí, un ejemplo del código para criterio ascendiente:

```
lista =  [random.randint(1, 1000) for _ in range(10000)]
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 ]
```

Gráficamente esto sería así (supuesto: ordenamiento ascendente): lista=[6, 4, 3, 1, 5]

| n=5 | | | | | |
|---|---|---|---|---|---|
| vuelta 1  i=0 / j=0 a 3 | 6 4 3 1 5 | 4 6 3 1 5 | 4 3 6 1 5 | 4 3 1 6 5 | 4 3 1 5 6 |
| vuelta 2 i=1 / j=0 a 2 | 4 3 1 5 6 | 3 4 1 5 6 | 3 1 4 5 6 | 3 1 4 5 6 | |
| vuelta 3 i=2 / j=0 a 1 | 3 1 4 5 6 | 1 3 4 5 6 | 1 3 4 5 6 | | |
| vuelta  i=3 / j=0 | 1 3 4 5 6 | 1 3 4 5 6 | | | |
| 5ta. vuelta  i=4 | 1 3 4 5 6 | | | | |

En el siguiente enlace podrán observar una animación del proceso: https://tute-avalos.com/images/bubblesort.gif

**2. Ordenamiento por Selección (Selection Sort).**

Este método funciona encontrando el elemento más pequeño de la lista y luego intercambiándolo con el primer elemento. Este proceso se repite hasta que todos los elementos de la lista están ordenados. Es decir, toma el primer elemento de la lista y lo va comparando con todos los demás elementos y va registrando en que posición está el elemento que es menor o mayor (dependiendo el criterio establecido). Si encuentra uno menor o mayor intercambian las posiciones y repite el procedimiento con el segundo, tercer, n, elemento.

Mostramos aquí, un ejemplo del código para criterio ascendente:

```
n = len(lista)
    for i in range(n):
        # Encontrar el índice del elemento mínimo
        min_index = i
        for j in range(i+1, n):
            if lista[ j ] < lista[ min_index ]:
                min_ index = j
        # Intercambiar el elemento mínimo con el elemento actual
        lista[ i ], lista[ min_index ] = lista[ min_index ], lista[ i ]
```

La ventaja de este método radica en la simple implementación y estabilidad[1] en el número de intercambios, y su desventaja es la ineficiencia para listas grandes.

### 3. Ordenamiento por Inserción (Insertion Sort).

Este método recorre el vector y toma el valor evaluado, lo compara con cada uno de los elementos anteriores hasta insertarlo en el lugar correcto.

Mostramos aquí, un ejemplo del código para criterio ascendente:

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

Cómo funciona: toma el primer elemento y lo compara con los elementos que están a su derecha. Si cumple la condición establecida, se va reordenando hasta que ya no se cumple la condición. Este proceso se repite hasta que se analiza el último elemento de la lista.

[1] garantiza que no cambia el orden relativo de elementos que resultan igual en la. comparación — esto es de gran ayuda para ordenar en múltiples pases

En el siguiente enlace podrán observar una animación del proceso: https://tute-avalos.com/images/insertionsort.gif

### 4. Ordenamiento rápido o Quicksort

Este algoritmo está basado en la técnica de divide y vencerás. Se toma el primer elemento como pivote, y se recorren todos los elementos comparando con el primero, armando dos sublistas, una contendrá todos los elementos menores al pivote y la otra los mayores, pero todavía desordenada. Este proceso se repite con cada sublista hasta que queda un o ningún elemento en ellas y se empieza a formar la lista final con cada una de las sublistas que están ordenadas.

Mostramos aquí, un ejemplo del código para criterio ascendente:

```
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivot = lista[0]
        menor = [x for x in lista[1:] if x <= pivot]
        mayor = [x for x in lista[1:] if x > pivot]
        return quicksort(menor) + [pivot] + quicksort(mayor)
```

Este método, implementado en forma recursiva, puede generar un error en python de tipo “RecursionError: maximum recursion depth exceeded in comparison” para listas muy grandes. Como solución, se puede implementar de forma iterativa o utilizar la función sorted() de python que incluye métodos como Timsort que no abordaremos en este trabajo.

Mostramos aquí, un ejemplo del código iterativo, para criterio ascendente:

```
def quick_sort_iterativo(lista):
    if len(lista) <= 1:
        return lista

    stack = [(0, len(lista) - 1)]

    while stack:
        start, end = stack.pop()

        if start >= end:
```

```
            continue

        pivot = lista[end]
        i = start

        for j in range(start, end):
            if lista[ j ] < pivot:
                lista[ i ], lista [ j ] = lista[ j ], lista [ i ]
                i += 1

        lista[ i ], lista[ end ] = lista[ end ], lista[ i ]

        stack.append((start, i - 1))
        stack.append((i + 1, end))

    return lista
```

Esta es una versión in-place (en el lugar) que usa el último elemento como pivote, y reordena la lista sin dividirla en sublistas. Técnicamente es QuickSort, pero in-place comúnmente usada cuando se quiere eficiencia en espacio. Usa índices y una pila simulada para evitar recursión y reducir consumo de memoria. Manipula la lista en el mismo espacio de memoria, evitando crear nuevas listas, que es costoso en listas grandes.

Exponemos aquí un cuadro comparativo relativo a la velocidad y uso de memoria de cada método.

| Característica | Burbuja | Selección | Inserción | Quicksort Recursivo | Quicksort Iterativo |
|---|---|---|---|---|---|
| Tiempo (Peor Caso) | O(n2) | O(n2) | O(n2) | O(n2) | O(n2) |
| Tiempo (Promedio) | O(n2) | O(n2) | O(n2) | O(nlogn) | O(nlogn) |
| Tiempo (Mejor Caso) | O(n) | O(n2) | O(n) | O(nlogn) | O(nlogn) |
| Uso de Memoria | O(1) (In-place) | O(1) (In-place) | O(1) (In-place) | O(logn) | O(logn) |
| Estabilidad | Sí | No | Sí | No | No |

| Tipo de Ordenamiento | Comparación | Comparación | Comparación | Comparación | Comparación |
|---|---|---|---|---|---|
| Consideracio-nes | Muy lento para grandes conjuntos de datos | Mejor para pequeños conjuntos de datos | Eficiente para conjuntos casi ordenados | Muy rápido en promedio, susceptible al peor caso | Evita la recursión, control de desbordamiento de pila |

## Búsqueda

¿Para qué sirven los algoritmos de búsqueda?

Los algoritmos de búsqueda son operaciones fundamentales en programación que utilizamos para encontrar un elemento específico dentro de un conjunto de datos.Por ejemplo una base de datos con millones de registros si no tenemos un algoritmo de búsqueda , sería difícil encontrar un dato.

Estos algoritmos suelen usarse para:

- Bases de datos**:** Encontrar registros de personas, productos, etc.
- Motores de búsqueda**:** Localizar páginas web relevantes.
- Inteligencia Artificial**:** Planificación de rutas, resolución de problemas complejos.
- Videojuegos**:** Movimiento de personajes, búsqueda de caminos.
- Análisis de datos**:** Encontrar patrones o valores específicos.
- Robótica**:** Navegación y planificación de trayectorias.

**1.** Búsqueda Lineal (o Secuencial):

- Funcionamiento: Recorre cada elemento de la lista o arreglo uno por uno, desde el principio hasta el final, hasta encontrar el valor deseado o hasta que se haya revisado toda la lista, este tipo de búsqueda lineal no necesita que los datos estén ordenados para funcionar, por lo que se utiliza principalmente en conjuntos de datos sin ordenar.
- Ventajas: Fácil de implementar, funciona en listas no ordenadas.Esto se vuelve útil en situaciones en las que no es práctico ordenar, o cuando trabajamos con datos en bruto
- Desventajas: Muy ineficiente para listas grandes, ya que en el peor de los casos, necesita revisar todos los elementos. Su complejidad temporal es O(n),lo que significa que puede tener que comprobar cada elemento en el peor de los casos.

Existen dos tipos de métodos para realizar una búsqueda lineal, el Método iterativo y el método recursivo.:

Búsqueda Lineal Iterativa

- Ventajas:
  - Simplicidad: Es muy fácil de entender e implementar.
  - Eficiencia en memoria: Solo necesita unas pocas variables para funcionar, por lo que consume muy poca memoria adicional (O(1)).
  - Sin riesgo de desbordamiento de pila: No hay problemas con la cantidad de elementos en la lista, ya que no usa la pila de llamadas para el proceso de búsqueda.
- Desventajas:
  - Velocidad: Es lenta para listas muy grandes, ya que en el peor de los casos (o si el elemento no está) tiene que revisar cada elemento. Su complejidad es O(n).

Este es un ejemplo gráfico de cómo funciona este tipo de búsqueda:

Designamos los valores de una lista con sus respectivos valores:

| Índices | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Valores | 1 | 3 | 7 | 9 | 13 | 19 | 21 |

Si el valor que queremos buscar es el 19 el algoritmo irá comparando con cada elemento uno por uno, comenzando desde el índice 0 hasta encontrar el valor designado

| Índices | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Valores | 1 | 3 | 7 | 9 | 13 | 19 | 21 |

Este es el paso a paso de cómo se ejecutaría la búsqueda:

1: 1 ≠ 19 Paso 2: 3 ≠ 19 Paso 3: 7 ≠ 19 Paso 4: 9 ≠ 19 Paso 5: 13 ≠ 19 Paso 6: 19 = 19

En el paso 6 se encontró el valor 19

A continuación mostramos un ejemplo de código:

```
def busqueda_lineal(lista, valor_objetivo):
    print(f"Valor Objetivo: {valor_objetivo}")

    for i in range(len(lista)):
        print(f"Paso {i + 1}:")
        print(f"  Comparando {lista[i]} con {valor_objetivo}.")
        if lista[i] == valor_objetivo:
            print(f"  {lista[i]} = {valor_objetivo}")
            print("¡Éxito!")
            return i
        else:
            print(f"  {lista[i]} ≠ {valor_objetivo}")
            print(f"  No es igual, pasando al siguiente elemento.")

    print("El valor objetivo no fue encontrado en la lista.")
    return -1

valores = [1, 3, 7, 9, 13, 19, 21, 31, 41]
target = 19

indice_encontrado = busqueda_lineal(valores, target)

if indice_encontrado != -1:
    print(f"\nEl valor {target} fue encontrado en el índice: {indice_encontrado}")
else:
    print(f"\nEl valor {target} no se encontró en la lista.")
```

Funciona de la siguiente manera:

1. Iteración secuencial: Recorre la lista elemento por elemento, desde el principio hasta el final.
2. Comparación**:** En cada paso, compara el elemento actual de la lista con el valor_objetivo.
3. Éxito**:** Si el elemento actual es igual al valor_objetivo, la búsqueda se detiene y devuelve el índice (posición) de ese elemento.
4. Fallo**:** Si se recorre toda la lista y el valor_objetivo no se encuentra, la función indica que no fue hallado (generalmente devolviendo -1).

2. Búsqueda Binaria:

- Funcionamiento: Este algoritmo es mucho más eficiente que la búsqueda lineal, pero requiere que la lista o el arreglo está ordenado. Divide repetidamente el intervalo de búsqueda por la mitad. Compara el valor buscado con el elemento del medio; si no son iguales, descarta la mitad en la que el valor no puede estar y continúa la búsqueda en la mitad restante.
- Ventajas: Muy eficiente para listas grandes y ordenadas. Su complejidad temporal es O(logn).
- Desventajas**:** Solo funciona en datos ordenados. Si los datos no están ordenados, primero se necesitaría aplicar un algoritmo de ordenación.

Al igual que en la busqueda línea , existen dos métodos para realizar una búsqueda binaria , el método iterativo y el método recursivo

### Búsqueda Binaria Iterativa

- Ventajas:
  - Eficiencia en espacio**:** Consume una cantidad constante de memoria (O(1)) porque solo necesita unas pocas variables. No usa la "pila de llamadas" (call stack) del programa.
  - Rendimiento**:** Generalmente es un poco más rápida en la práctica porque evita la sobrecarga de las llamadas a funciones.
  - Sin riesgo de desbordamiento de pila**:** No hay peligro de que el programa se caiga por recursión excesiva en listas muy grandes.
- Desventajas**:**
  - El código puede parecer un poco menos intuitivo o "elegante" al principio, ya que requiere gestionar los límites del bucle manualmente.

### Búsqueda Binaria Recursiva

- Ventajas:
  - Claridad**:** Para problemas que son inherentemente recursivos (como "divide y vencerás"), el código recursivo puede ser más conciso y fácil de leer para algunas personas.
- Desventajas:
  - Mayor consumo de espacio: Cada llamada recursiva añade un "marco" a la pila de llamadas del programa. Para listas muy grandes, esto puede llevar a un desbordamiento de pila (stack overflow) y la complejidad espacial es O(logn).


- Rendimiento: Puede ser marginalmente más lenta debido a la sobrecarga de gestionar la pila de llamadas.

Método iterativo

Ejemplo gráfico:

Valor Objetivo: Se busca el número 31 en la lista.

Primer Paso:

La lista inicial va del índice 0 al 8.

Se calcula el índice medio: (0+8)/2=4.

Se compara el valor objetivo (31) con el valor en el índice medio (que es 13). Como 13 es menor que 31, se sabe que el valor objetivo debe estar en la mitad superior de la lista.

| Índices | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Valores | 1 | 3 | 7 | 9 | 13 | 19 | 21 | 31 | 41 |

Segundo Paso:

- Ahora, la búsqueda se restringe a la mitad superior de la lista (del índice 5 al 8).
- Se calcula el nuevo índice medio: (5+8)/2=6 (redondeado hacia abajo).
- Se compara el valor objetivo (31) con el valor en el índice 6 (que es 21). Como 21 es menor que 31, se sabe que el valor objetivo debe estar en la mitad superior restante.

| Índices | 5 | 6 | 7 | 8 |
|---|---|---|---|---|
| Valores | 19 | 21 | 31 | 41 |

Tercer Paso:

- La búsqueda se restringe aún más, ahora a los índices 7 y 8.
- Se calcula el nuevo índice medio: (7+8)/2=7 (redondeado hacia abajo).
- Se compara el valor objetivo (31) con el valor en el índice 7 (que es 31)

| | | |
|---|---|---|
| Índices | 7 | 8 |
| Valores | 31 | 41 |

Encontramos el valor que estábamos buscando:

- El valor 31 es igual al valor en el índice 7, por lo tanto,la búsqueda termina con éxito.

Código en python de el ejemplo desarrollado:

```
def busqueda_binaria(lista_ordenada, valor_objetivo):
    bajo = 0
    alto = len(lista_ordenada) - 1

    while bajo <= alto:
        medio = (bajo + alto) // 2
        valor_en_medio = lista_ordenada[medio]

        if valor_en_medio == valor_objetivo:
            return medio
        elif valor_objetivo < valor_en_medio:
            alto = medio - 1
        else:
            bajo = medio + 1

    return -1

valores_ordenados = [1, 3, 7, 9, 13, 19, 21, 31, 41]
target = 31

indice_encontrado = busqueda_binaria(valores_ordenados, target)

if indice_encontrado != -1:
    print(f"El valor {target} fue encontrado en el índice: {indice_encontrado}")
else:
    print(f"El valor {target} no se encontró en la lista.")
print("\n--- Probando con un valor no existente ---")
target_no_existente = 100
indice_no_encontrado = busqueda_binaria(valores_ordenados, target_no_existente)

if indice_no_encontrado != -1:
```


```
        print(f"El valor {target_no_existente} fue encontrado en el índice: {indice_no_encontrado}")
else:
   print(f"El valor {target_no_existente} no se encontró en la lista.")
```

En las últimas 8 líneas se realizó una comprobación en caso de que el valor no se haya encontrado, el valor solicitado en esta prueba fue target_no_existente = 100

Cómo funciona

Recordemos siempre que este código busca un **v**alor específico en una lista que DEBE ESTAR ORDENADA.

Funciona así:

1. Define un rango: Empieza buscando en toda la lista, usando bajo (inicio) y alto (fin).
2. Encuentra el medio: Calcula el elemento del medio dentro de ese rango.
3. Compara:
   - Si el valor del medio es el que buscas, ¡lo encontraste y devuelve su posición!
   - Si el valor que buscas es menor que el del medio, entonces el elemento debe estar en la mitad izquierda. El código descarta la mitad derecha ajustando alto.
   - Si el valor que buscas es mayor que el del medio, entonces el elemento debe estar en la mitad derecha. El código descarta la mitad izquierda ajustando bajo.
4. Repite**:** Vuelve al paso 2 con el nuevo rango reducido, dividiéndolo a la mitad otra vez.
5. No encontrado: Si los límites bajo y alto se cruzan (ya no queda rango para buscar), significa que el valor no está en la lista y devuelve -1.

Método Recursivo

A diferencia de la versión iterativa que usa un bucle while, la recursiva se llama a sí misma repetidamente para dividir la lista.

```
def busqueda_binaria_recursiva(lista_ordenada, valor_objetivo, bajo, alto):
```


```
    if bajo > alto:
        return -1

    medio = (bajo + alto) // 2
    valor_en_medio = lista_ordenada[medio]

    if valor_en_medio == valor_objetivo:
        return medio
    elif valor_objetivo < valor_en_medio:
         return busqueda_binaria_recursiva(lista_ordenada, valor_objetivo, bajo, medio -
1)
    else:
          return busqueda_binaria_recursiva(lista_ordenada, valor_objetivo, medio + 1,
alto)

valores_ordenados = [1, 3, 7, 9, 13, 19, 21, 31, 41]
target = 31

indice_encontrado = busqueda_binaria_recursiva(valores_ordenados, target, 0,
len(valores_ordenados) - 1)

if indice_encontrado != -1:
    print(f"El valor {target} fue encontrado en el índice: {indice_encontrado}")
else:
    print(f"El valor {target} no se encontró en la lista.")

target_no_existente = 100
indice_no_encontrado = busqueda_binaria_recursiva(valores_ordenados,
target_no_existente, 0, len(valores_ordenados) - 1)

if indice_no_encontrado != -1:
          print(f"El valor {target_no_existente} fue encontrado en el índice:
{indice_no_encontrado}")
else:
    print(f"El valor {target_no_existente} no se encontró en la lista.")
```

## 3. Caso Práctico

Realizamos un algoritmo en Python para comparar los tiempos de ejecución de cada uno de los métodos de ordenamiento y búsqueda tratados, en base a las mismas


listas, probando con una lista de personas de 2700 elementos aproximadamente y una lista de números pequeña y otra aleatoria pero más grande.

En el caso de la lista aleatoria y para evitar diferencias por enfrentar distintas listas, la lista que evalúa el primer algoritmo (Método burbuja), se copia en otras listas que son procesadas por el resto de los algoritmos.

## 4. Metodología Utilizada

Para la confección de trabajo se efectuaron las siguientes etapas:

1.- Recopilación de información teórica.

2.- Estudio de todo el material teórico provisto por la cátedra de Programación I de la Tecnicatura Universitaria en Programación a Distancia dictada por la UTN.

3.- Visualización del material audiovisual provisto por la citada cátedra.

4.- Análisis y comprensión de cada uno de los algoritmos propuestos, en función a la metodología elegida (búsqueda lineal o binaria; ordenamiento por burbuja, por selección, por inserción y quicksort en sus versiones recursivas e iterativas.)

5.- Desarrollo, implementación y prueba en Python de los algoritmos analizados.

6.- Registro de los resultados y validación de funcionalidad

7.- Elaboración del informe

## 5. Resultados Obtenidos

Metodos de ordenamiento:

- Por burbuja (bubble sort): se logró implementar correctamente para listas pequeñas debido a que en un rango de lista muy grande se vuelve ineficiente
- Por Selección (Selection Sort**):** se desarrolló y resulto útil en listas pequeñas pero no en grandes
- Por Inserción (Insertion Sort): se implementó de manera correcta pero presentó problemas de rendimiento para listas grandes debido a su complejidad temporal cuadrática.
- Por Ordenamiento rápido o Quicksort: presentó problemas para ejecutar de manera recursiva debido los errores de desbordamiento de pila pero en su forma iterativa resultó eficiente

Metodos de busqueda:

- El método lineal iterativo : se logró implementar correctamente demostrando la utilidad y sencillez para buscar valores en listas no ordenadas
- La búsqueda binaria iterativa :se ejecutó correctamente demostrando que es una manera más rápida para realizar búsquedas con el requisito de que las listas deben si o si estar ordenadas
- La búsqueda binaria por el método recursivo: se logró implementar sin problemas y, a diferencia del método lineal iterativo no presentó errores debido al exceso de llamadas permitidas a las funciones

## 6. Conclusiones

A través de este trabajo logramos desarrollar e implementar algoritmos de búsqueda y ordenamiento que se ejecutan correctamente según las necesidades,, ya sea para buscar datos de manera ordenada o desordenada, o para realizar búsquedas en listas grandes o pequeñas.También se implementaron algoritmos para generar listas pequeñas y grandes junto con funciones para medir el tiempo de respuesta de cada algoritmo.

Los algoritmos de burbuja, selección e inserción fueron ineficientes para grandes listas. Sin embargo, tienen la ventaja de no requerir uso de memoria adicional y, en el caso de burbuja e inserción, son estables.

Quicksort, tanto en su versión recursiva como iterativa, fueron más eficientes en promedio. Esto lo convierte en una mejor opción para ordenar grandes listas.

La principal diferencia entre Quicksort recursivo e iterativo radica en cómo gestionan la pila de llamadas. La versión recursiva utiliza la pila de llamadas del sistema operativo, lo que puede llevar a un desbordamiento de pila para conjuntos de datos extremadamente grandes. La versión iterativa gestiona su propia pila explícitamente, lo que permite un mayor control y evita este problema.

También aprendimos acerca de los errores que pueden surgir al ejecutar los algoritmos en listas de gran tamaño y cómo resolver las mismas.

## 7. Bibliografía


- Material de estudio de la Cátedra Programación I de la carrera Tecnicatura Universitaria en Programación a distancia, dictada por la Universidad Tecnológica Nacional.
- **"Apunte No 10. Búsqueda y Ordenamiento" del Prof. Tute Ávalos:**
- https://tute-avalos.com/static/python/10%20-%20Algoritmos%20de%20b%C3%BAsqueda%20y%20ordenamiento.pdf.
- Documentación oficial de Python:
    - https://python-docs-es

- https://www.datacamp.com/es/tutorial/linear-search-python
- https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

## 8. Anexos

1.- Tiempo de respuesta Algoritmos de Ordenamiento

2.- Tiempo de respuesta de Algoritmos de búsqueda.

3. Repositorio

https://github.com/juanegelos/TP_INTEG_PROGRAMACION_TUPaD.git

4. Video: https://www.youtube.com/watch?v=5scjcICuElw