Trabajo Integrador – Programación I

Datos Generales

- Título del trabajo: Algoritmos de Búsqueda y Ordenamiento en Python

- Alumnos: Iván Nievas Zorn, Alexis Javier Pajón Fenoglio

- Materia: Programación I

- Profesora: Cinthia Rigoni

- Fecha de Entrega: 09/06/2025

Índice

1. Introducción

En el presente trabajo se abordan distintos algoritmos de búsqueda y ordenamiento utilizando el lenguaje de programación Python. El propósito es analizar su funcionamiento, eficiencia y aplicabilidad a través de un caso práctico concreto.

El tema fue seleccionado por su relevancia dentro de la programación, ya que estos algoritmos constituyen la base de casi cualquier programa.

El objetivo que nos proponemos es comparar sus rendimientos en listas de datos reales y preparar dicha información para posteriores búsquedas eficientes.

2. Marco Teórico

Ordenamiento

Los algoritmos de ordenamiento son procesos que reordenan los elementos de una lista o arreglo, siguiendo un criterio determinado. El criterio más común es el numérico ascendente o descendente, aunque también puede aplicarse a datos alfabéticos, cronológicos, entre otros. Elegir el correcto es determinante ya que muchas veces es necesario contar con los datos ordenados para realizar búsquedas de manera eficiente. Además, esta elección impacta directamente en el rendimiento del programa, especialmente al trabajar con grandes volúmenes de datos.

Tipos de ordenamientos:

- **Insertion Sort**

  El algoritmo recorre la lista desde el segundo elemento y va insertando cada valor en la posición correcta de la parte ya ordenada.

```python
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
    return arr
```

Complejidad: O(n²) en el peor caso
Ventajas: Bueno para listas pequeñas o casi ordenadas
Desventajas: Lento con listas grandes

- ## Selection Sort

Busca el mínimo de la lista no ordenada y lo intercambia con el primer elemento, y así sucesivamente.

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr
```

Complejidad: O(n²) en todos los casos
Ventajas: Fácil de entender e implementar
Desventajas: Ineficiente para listas grandes

- ## Quick Sort

Divide la lista en dos sublistas usando un pivote, y ordena cada parte de forma recursiva.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

Complejidad promedio: O(n log n)
Peor caso: O(n²)
Ventajas: Rápido en la mayoría de los casos
Desventajas: No es estable y puede consumir más memoria

- ### sort()

  Método interno de listas que modifica la lista original.

```
def usar_sort(lista):
    lista.sort()
    return lista
```

Complejidad: O(n log n)
Ventajas: Muy eficiente
Desventajas: Modifica la lista original

- ### sorted()

  Función que devuelve una nueva lista ordenada, sin cambiar la original.

```
def usar_sorted(lista):
    return sorted(lista)
```

Complejidad: O(n log n)
Ventajas: No altera la lista original
Desventajas: Usa más memoria

Búsqueda

Un algoritmo de búsqueda es una secuencia de pasos lógicos que permite localizar un dato específico dentro de una estructura, como una lista. Existen distintos tipos de algoritmos de búsqueda, y su elección depende de factores como el tamaño de la lista, el orden de los elementos y la eficiencia deseada.

Tipos de búsqueda:

- ### Búsqueda lineal

También llamada búsqueda secuencial, es el algoritmo más simple. Consiste en recorrer uno por uno los elementos de la lista desde el inicio hasta encontrar el valor deseado o hasta llegar al final de la lista.

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

Ventajas:

- No requiere que la lista esté ordenada.
- Es fácil de implementar.

Desventajas:

- Su eficiencia disminuye en listas grandes

Complejidad Temporal:

- Mejor caso: O(1) = el elemento está al principio.
- Peor caso: O(n) = el elemento está al final o directamente no está.
- Caso promedio: O(n) = el elemento está a la mitad o cerca.

- ## Búsqueda binaria

Es un algoritmo más eficiente, pero requiere que la lista esté previamente ordenada. Su funcionamiento se basa en dividir repetidamente la lista por la mitad y comparar el valor medio con el objetivo, descartando la mitad que no contiene el dato.

```python
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
```

```
        else:
            fin = medio - 1
    return -1
```

Ventajas:

- Muy rápida en listas grandes.
- Reduce drásticamente la cantidad de comparaciones,de esta manera ahorrando recursos.

Desventajas:

- Solo funciona en listas ordenadas
- Es un poco más compleja de implementar

Complejidad Temporal:

- Mejor caso: O(1) = el objetivo está justo en el medio.
- Peor caso: O(log n) = el objetivo se encuentra en la última iteración o no está.
- Caso promedio: O(n) = se deben realizar cierta cantidad de iteraciones para encontrar el objetivo.

3. Caso Práctico

Algoritmo de ordenamiento

- Descripción del problema abordado:

El problema consiste en comparar los tiempos de ejecución de diferentes algoritmos de ordenamiento sobre una lista de datos reales. Para esto, se utilizaron las poblaciones de 331 ciudades argentinas extraídas de la base de datos "Argentina Cities Database" de SimpleMaps (SimpleMaps, n.d.). La finalidad es analizar el comportamiento de varios algoritmos de ordenamiento (Insertion Sort, Selection Sort, Quick Sort, sort() y sorted()) al trabajar con esta lista y evaluar cuál es el más eficiente en términos de tiempo de ejecución.

- Código fuente comentado:

```python
# Este script compara el rendimiento de distintos algoritmos de ordenamiento

# sobre una lista grande y otra reducida de poblaciones argentinas.



import random

import timeit



print("="*60)

print("Comparación de Algoritmos de Ordenamiento en Python".center(60))

print("="*60)



# Lista mezclada de poblaciones de las 331 ciudades mas pobladas de
Argentina

poblaciones_arg = [

    21624, 730266, 24307, 15741, 39601, 37745, 58315, 82227, 296826, 7983,
121451, 6351,

    22924, 43251, 47839, 85060, 49132, 3812, 63284, 79476, 352646, 115353,
64556, 48140,

    180995, 28537, 26651, 405683, 99066, 43287, 34421, 11428, 17491, 46421,
8946, 7902,

    19142, 4570, 30903, 27000, 57244, 106899, 231198, 42276, 109461, 81465,
244168, 25757,

    33607, 291720, 11785, 31901, 64867, 160954, 247863, 13689, 47622, 5465,
12029, 157532,

    52529, 85189, 43009, 58430, 26221, 50426, 9294, 12537, 163240, 2106734,
65859, 106441,

    8260, 18349, 169947, 15394, 32097, 57669, 32116, 16710000, 41705, 903,
40259, 400,
```

29205, 15126, 301572, 55728, 10873, 32646, 54698, 91322, 26497, 46239, 12801, 50183,

120346, 38000, 5547, 12621, 30680, 44796, 41109, 160000, 11562, 10067, 12799, 92945,

57323, 7591, 47348, 159139, 62315, 45848, 31602, 75315, 9012, 7772, 25705, 14692, 54081,

73058, 32027, 215020, 56419, 90000, 223898, 98859, 41403, 56956, 48158, 7110, 23408,

275988, 104985, 43595, 3317, 107786, 95785, 160219, 80769, 72304, 87258, 200000, 29854,

190696, 35307, 47374, 27004, 10885, 147965, 98017, 24747, 10902, 321789, 35465, 46642,

100324, 31553, 481, 66112, 262379, 191299, 59031, 29593, 471389, 47626, 75616, 11101,

31892, 76070, 107778, 6084, 21997, 89882, 60867, 38470, 11589, 459263, 216637, 28051,

41463, 54678, 112580, 35301, 44529, 6916, 54463, 65881, 50057, 19877, 63226, 106662,

6320, 149450, 23150, 16874, 88470, 23171, 9244, 52837, 22953, 47569, 25520, 25940, 136604,

82582, 10470, 85315, 58152, 24616, 29218, 112887, 42082, 27956, 605767, 535303, 238067,

100728, 57458, 41390, 85420, 62001, 30100, 11316, 23299, 12733, 41155, 31758, 4569,

7652, 65684, 9763, 32083, 18089, 88600, 36994, 21333, 89721, 49782, 2064, 180523, 682605,

32097, 72000, 92957, 26448, 22046, 60000, 21125, 57878, 10889, 95796, 39507, 73293,

30028, 21592, 86860, 37775, 25140, 18115, 15347, 102880, 33724, 17634, 35058, 115041,

```
    46019, 28750, 2139, 28814, 63196, 135275, 64640, 6249, 82582, 28339,
234000, 10743, 3973,

    73155, 43168, 33222, 13462, 35844, 36494, 25404, 13245, 911506, 109914,
109644, 4429,

    13516, 31190, 10319, 51448, 58811, 23470, 71479, 21088, 78699, 94403,
5534, 193144, 41176,

    39151, 23274, 97915, 59335, 22824, 38418, 1276000, 49445, 16658, 79557,
23027, 111391,

    146704, 98522, 32390, 62000, 33835, 28265, 8351, 11672, 40858, 30666,
189067, 32227, 971,

    60752

]



total_ciudades = len(poblaciones_arg)

print(f"La lista sin ordenar tiene las poblaciones de un total de
{total_ciudades} ciudades")



# Ordenamos inversamente la lista poblaciones_arg para tomar las 10
poblaciones mas altas.

poblaciones_arg_top10 = sorted(poblaciones_arg, reverse=True)[:10]



# Algoritmos

# Funcion Insertion sort

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
    return arr


# Funcion selection sort
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr


# Funcion Quicksort
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
```

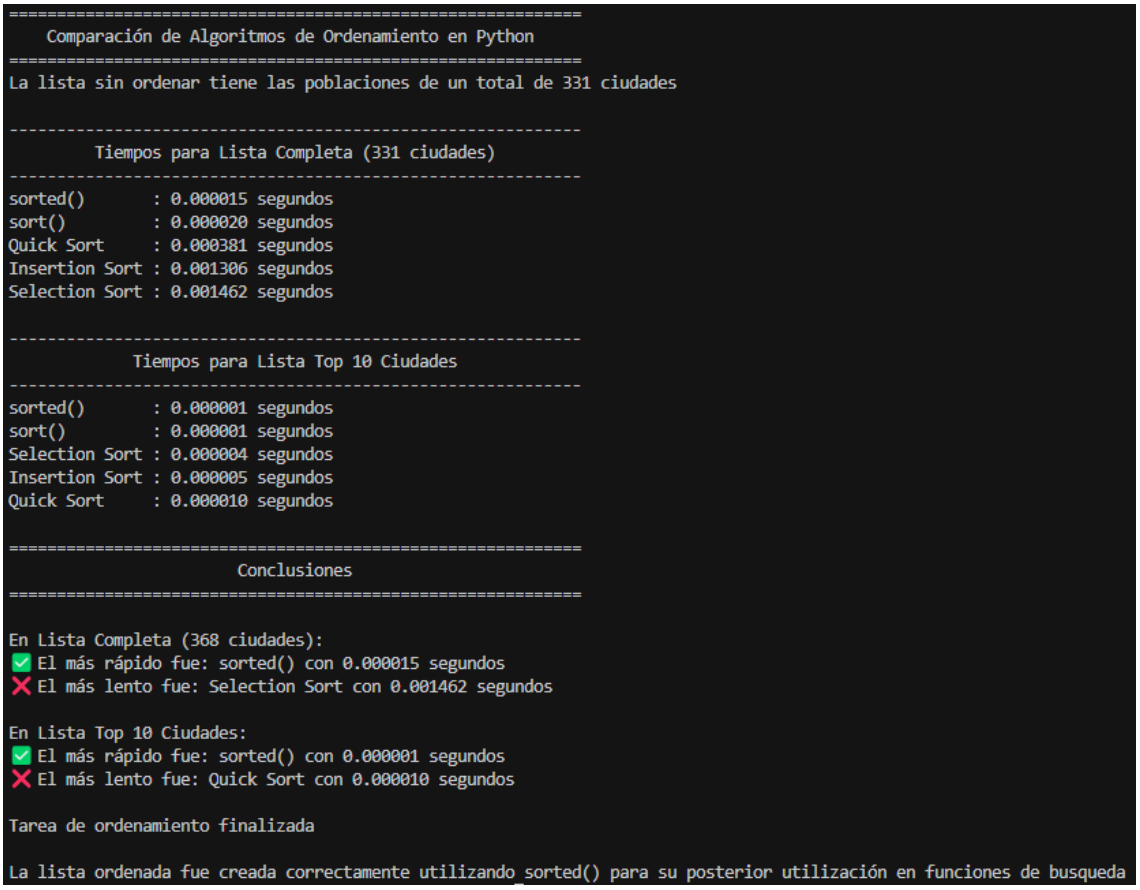
```
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)


# Funciones sort()
def usar_sort(lista):
    lista.sort()
    return lista


# Funcion sorted()
def usar_sorted(lista):
    return sorted(lista)


# Función para medir tiempo de ejecución
def medir_tiempo(func, lista):
    inicio = timeit.default_timer()
    func(lista.copy())
    fin = timeit.default_timer()
    return fin - inicio


def mostrar_resultado(nombre_lista, mejor, peor, tiempos):
    print(f"\nEn {nombre_lista}:")
```

```
    print(f"✅ El más rápido fue: {mejor} con {tiempos[mejor]:.6f}
segundos")

    print(f"❌ El más lento fue: {peor} con {tiempos[peor]:.6f} segundos")



# Crear un diccionario vacío para guardar los tiempos

tiempos_lista_completa = {}

tiempos_lista_top10 = {}



# Diccionario unificado de algoritmos con nombres consistentes

algoritmos = {

    "Insertion Sort": insertion_sort,

    "Selection Sort": selection_sort,

    "Quick Sort": quicksort,

    "sort()": usar_sort,

    "sorted()": usar_sorted

}



for metodo, funcion in algoritmos.items():

    tiempos_lista_completa[f"{metodo}"] = medir_tiempo(funcion,
poblaciones_arg)

    tiempos_lista_top10[f"{metodo}"] = medir_tiempo(funcion,
poblaciones_arg_top10)
```

```
# Mostrar resultados ordenados. # Ordenamos las claves del diccionario
"tiempos", utilizando el valor asociado a cada clave (tiempo)
key=tiempos.get, de menor a mayor.

print("\n" + "-"*60)

print(f"Tiempos para Lista Completa ({total_ciudades} ciudades)".center(60))

print("-"*60)

for metodo in sorted(tiempos_lista_completa,
key=tiempos_lista_completa.get):

    print(f"{metodo:15}: {tiempos_lista_completa[metodo]:.6f} segundos")
#":15" deja 25 espacios reservados para alinear los nombres a la izquierda.
".6f" muestra resultado con 6 decimales.



print("\n" + "-"*60)

print("Tiempos para Lista Top 10 Ciudades".center(60))

print("-"*60)

for metodo in sorted(tiempos_lista_top10, key=tiempos_lista_top10.get):

    print(f"{metodo:15}: {tiempos_lista_top10[metodo]:.6f} segundos") #":15"
deja 25 espacios reservados para alinear los nombres a la izquierda. ".6f"
muestra resultado con 6 decimales.



# Buscar el algoritmo más rápido y más lento para la lista completa de
ciudades.

# # key=tiempos.get le dice a min() o max() que, en lugar de comparar los
nombres, compare los valores del diccionario.

mejor_lista_completa = min(tiempos_lista_completa,
key=tiempos_lista_completa.get)

peor_lista_completa = max(tiempos_lista_completa,
key=tiempos_lista_completa.get)
```

```python
# Buscar el algoritmo más rápido y más lento para la lista reducida (10 ciudades más pobladas)

mejor_lista_top10 = min(tiempos_lista_top10, key=tiempos_lista_top10.get)

peor_lista_top10 = max(tiempos_lista_top10, key=tiempos_lista_top10.get)



print("\n" + "="*60)

print("Conclusiones".center(60))

print("="*60)

mostrar_resultado(f"Lista Completa ({total_ciudades} ciudades)", mejor_lista_completa, peor_lista_completa, tiempos_lista_completa)

mostrar_resultado("Lista Top 10 Ciudades", mejor_lista_top10, peor_lista_top10, tiempos_lista_top10)



# Preparamos la lista ordenada con el mejor algoritmo para que sea utilizada en la funcion de busqueda

lista_ordenada_para_busqueda = algoritmos[mejor_lista_completa](poblaciones_arg)

print(f"\nTarea de ordenamiento finalizada")

print(f"\nLa lista ordenada fue creada correctamente utilizando {mejor_lista_completa} para su posterior utilización en funciones de busqueda")
```

- - Captura de ejecución:



- Justificación de decisiones de diseño:

  Los algoritmos presentados por la cátedra fueron tres: Insertion Sort, Selection Sort y Quick Sort.

  Dado que se conocía de antemano el algoritmo sort(), se decidió agregarlo al listado de algoritmos. Además, durante la investigación, se descubrió el algoritmo sorted().

  Aunque existen otras opciones, se consideró que para ordenar datos numéricos, los cinco algoritmos utilizados son adecuados, según los resultados de la investigación realizada.

- Validación de los resultados obtenidos:

  De acuerdo a los resultados obtenidos, se concluyó que el método sorted() fue el más rápido, tanto al ordenar la lista completa como al ordenar la lista acotada.

  Las diferencias encontradas en cuanto al algoritmo más lento dependieron de si se trataba de una lista completa o acotada. En el caso de la lista completa, el algoritmo más lento fue Selection Sort, mientras que en el caso de la lista acotada, fue Quick Sort.

Algoritmo de búsqueda

Se decidió hacer una comparación entre la búsqueda lineal y binaria utilizando las listas anteriormente generadas, tanto ordenada como desordenada con el fin de demostrar en cual tipo de lista funcionan mejor y que tan eficiente es cada una.

- Código fuente comentado:

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


# Función búsqueda binaria
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1


    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            inicio = medio + 1
        else:
```

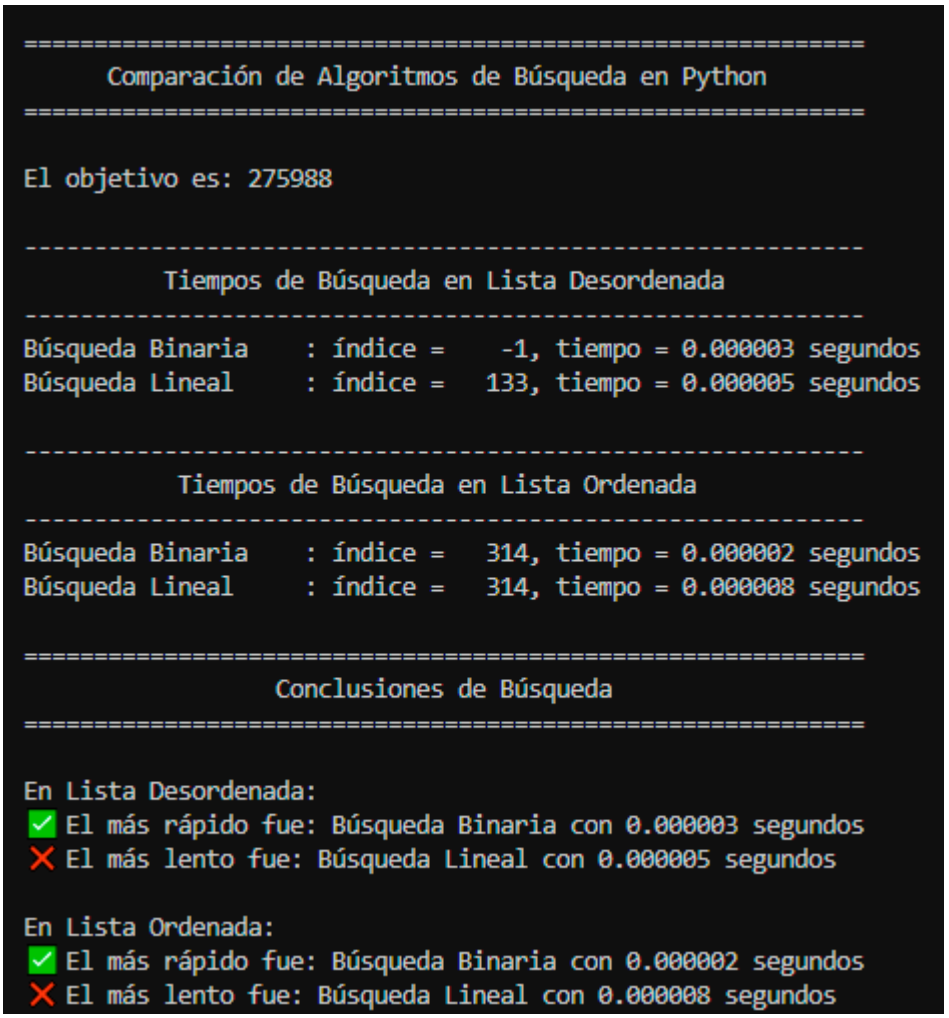
```
            fin = medio - 1
    return -1

# Función para medir el tiempo de búsqueda
def medir_tiempo_busqueda(func, lista, objetivo):
    inicio = timeit.default_timer()
    resultado = func(lista, objetivo)
    fin = timeit.default_timer()
    return resultado, fin - inicio

print("\n")
print("="*60)
print("Comparación de Algoritmos de Búsqueda en Python".center(60))
print("="*60)

objetivo = 275988
print(f"\nEl objetivo es: {objetivo}")

# Diccionario unificado
busquedas = {
    "Búsqueda Lineal": busqueda_lineal,
    "Búsqueda Binaria": busqueda_binaria
}
```

```
# Diccionarios para guardar tiempos e índices
indices_busqueda_desordenada = {}
tiempos_busqueda_desordenada = {}
indices_busqueda_ordenada = {}
tiempos_busqueda_ordenada = {}


for nombre, funcion in busquedas.items():
    indice, tiempo = medir_tiempo_busqueda(funcion, poblaciones_arg,
objetivo)
    tiempos_busqueda_desordenada[nombre] = tiempo
    indices_busqueda_desordenada[nombre] = indice


for nombre, funcion in busquedas.items():
    indice, tiempo = medir_tiempo_busqueda(funcion,
lista_ordenada_para_busqueda, objetivo)
    tiempos_busqueda_ordenada[nombre] = tiempo
    indices_busqueda_ordenada[nombre] = indice


print("\n" + "-"*60)
print("Tiempos de Búsqueda en Lista Desordenada".center(60))
print("-"*60)
for metodo in sorted(tiempos_busqueda_desordenada,
key=tiempos_busqueda_desordenada.get):
    tiempo = tiempos_busqueda_desordenada[metodo]
```

```
    indice = indices_busqueda_desordenada[metodo]

    print(f"{metodo:20}: índice = {indice:5}, tiempo = {tiempo:.6f}
segundos")



print("\n" + "-"*60)

print("Tiempos de Búsqueda en Lista Ordenada".center(60))

print("-"*60)

for metodo in sorted(tiempos_busqueda_ordenada,
key=tiempos_busqueda_ordenada.get):

    tiempo = tiempos_busqueda_ordenada[metodo]

    indice = indices_busqueda_ordenada[metodo]

    print(f"{metodo:20}: índice = {indice:5}, tiempo = {tiempo:.6f}
segundos")



mejor_des = min(tiempos_busqueda_desordenada,
key=tiempos_busqueda_desordenada.get)

peor_des = max(tiempos_busqueda_desordenada,
key=tiempos_busqueda_desordenada.get)

mejor_ord = min(tiempos_busqueda_ordenada,
key=tiempos_busqueda_ordenada.get)

peor_ord = max(tiempos_busqueda_ordenada, key=tiempos_busqueda_ordenada.get)



print("\n" + "="*60)

print("Conclusiones de Búsqueda".center(60))

print("="*60)
```

```
mostrar_resultado("Lista Desordenada", mejor_des, peor_des,
tiempos_busqueda_desordenada)

mostrar_resultado("Lista Ordenada", mejor_ord, peor_ord,
tiempos_busqueda_ordenada)
```

-Captura de ejecución:



- Justificación de decisiones de diseño:

Existen más tipos de búsqueda pero la cátedra hizo especial énfasis en la búsqueda lineal y binaria.

Los índices se añadieron para mostrar si el algoritmo de búsqueda efectivamente encontró el dato objetivo.

- Validación de los resultados obtenidos:

Según los anteriores resultados, el algoritmo de búsqueda binaria resultó ser el más eficiente en ambas listas, pero a diferencia del algoritmo de búsqueda lineal este solo encontró el objetivo en la lista ordenada mostrando su principal desventaja.

4. Metodología Utilizada

Describe los pasos seguidos para desarrollar el trabajo.

- Investigación teórica:

  El primer paso fue realizar una investigación teórica sobre los algoritmos de ordenamiento y búsqueda, con el fin de comprender cómo funcionan, qué complejidad tienen en los diferentes casos y qué ventajas o desventajas presentan. Esto incluyó el estudio de los algoritmos Insertion Sort, Selection Sort, Quick Sort, sort() y sorted() en Python.

  Se utilizó para ello: El Video sobre Búsqueda, Video sobre Ordenamiento y Notebook Búsqueda y Ordenamiento dentro de la plataforma de TUPaD. También se investigó alternativas recurriendo a w3schools para investigar sobre sort() y sorted() para obtener una comprensión del funcionamiento de cada algoritmo.

- Selección de datos:

  Para las pruebas de rendimiento, se seleccionó una lista de 331 ciudades argentinas, extraída de la base de datos Argentina Cities Database de SimpleMaps (SimpleMaps, n.d.). Esta base de datos contiene información sobre la población de las ciudades, lo que la hace adecuada para evaluar el rendimiento de los algoritmos de ordenamiento sobre datos reales.

- Implementación de los algoritmos:

  Se implementaron los cinco algoritmos de ordenamiento en Python, tal como se describió en el marco teórico. Para ello, se definieron funciones para cada uno de los algoritmos y se estructuraron de forma que pudieran ser fácilmente comparados en términos de tiempo de ejecución. Lo mismo aplica para los dos algoritmos de búsqueda añadiendo como extra la prueba de exactitud.

- Pruebas de rendimiento:

  Se realizaron pruebas con la lista de datos en dos condiciones diferentes:

  - Lista aleatoria (completamente desordenada)
  - Lista ordenada

  Para cada prueba, se midió el tiempo de ejecución de cada algoritmo utilizando la función time() de Python, que captura el tiempo antes y después de ejecutar cada algoritmo.

- Análisis de los resultados:

  Después de realizar las pruebas, se analizaron los tiempos de ejecución de cada algoritmo. Los resultados fueron comparados con la teoría sobre la eficiencia y la precisión de cada algoritmo, y se validó cómo los diferentes algoritmos se comportaron dependiendo de las características de la lista (Completa y acotada).

- Herramientas utilizadas: Python 3.11.9, Visual Studio Code 1.100.3, Argentina Cities Database de SimpleMaps (SimpleMaps, n.d.)
- Reparto de tareas entre los integrantes del grupo:
  - Iván Nievas Zorn se encargó del desarrollo y análisis de los algoritmos de ordenamiento (selection, insertion, quicksort, sort y sorted)
  - Alexis Javier Pajón Fenoglio se encargó del desarrollo y análisis de los algoritmos de búsqueda (lineal y binaria).
- Ambos colaboraron en la redacción del informe y las conclusiones generales.

5. Resultados Obtenidos

- Ejecuciones exitosas: Todos los algoritmos de ordenamiento implementados (Insertion Sort, Selection Sort, Quick Sort, sort() y sorted()) se ejecutaron correctamente sin errores. La ejecución de cada algoritmo sobre la lista de 331 ciudades argentinas se realizó de forma correcta, y se obtuvieron los resultados de los tiempos de ordenamiento para cada uno.

```
------------------------------------------------------------
        Tiempos para Lista Completa (331 ciudades)
------------------------------------------------------------
sorted()       : 0.000016 segundos
sort()         : 0.000020 segundos
Quick Sort     : 0.000366 segundos
Insertion Sort : 0.001432 segundos
Selection Sort : 0.001448 segundos

------------------------------------------------------------
            Tiempos para Lista Top 10 Ciudades
------------------------------------------------------------
sorted()       : 0.000001 segundos
sort()         : 0.000001 segundos
Selection Sort : 0.000004 segundos
Insertion Sort : 0.000004 segundos
Quick Sort     : 0.000010 segundos
```

De igual forma, los algoritmos de búsqueda se ejecutaron correctamente y se obtuvieron los resultados esperados con el único error (con fines demostrativos) del algoritmo de búsqueda binaria

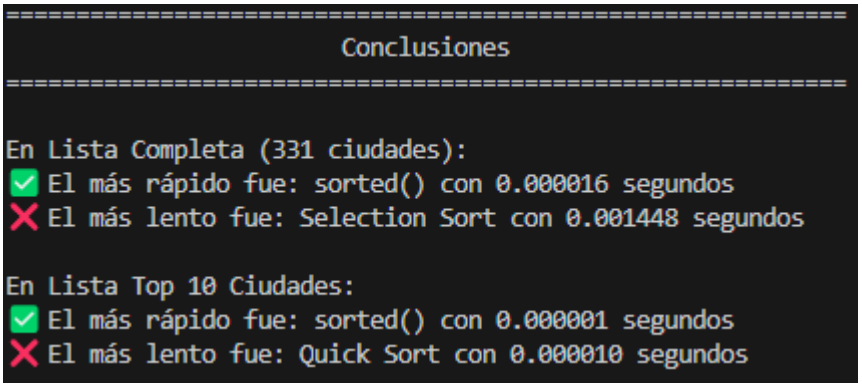
```
============================================================
      Comparación de Algoritmos de Búsqueda en Python
============================================================

El objetivo es: 275988

------------------------------------------------------------
          Tiempos de Búsqueda en Lista Desordenada
------------------------------------------------------------
Búsqueda Binaria    : índice =    -1, tiempo = 0.000003 segundos
Búsqueda Lineal     : índice =   133, tiempo = 0.000005 segundos

------------------------------------------------------------
           Tiempos de Búsqueda en Lista Ordenada
------------------------------------------------------------
Búsqueda Binaria    : índice =   314, tiempo = 0.000002 segundos
Búsqueda Lineal     : índice =   314, tiempo = 0.000008 segundos
```

- Captura de tiempos de ordenamiento y búsqueda: Se midió el tiempo de ejecución de cada algoritmo utilizando la función time() de Python.

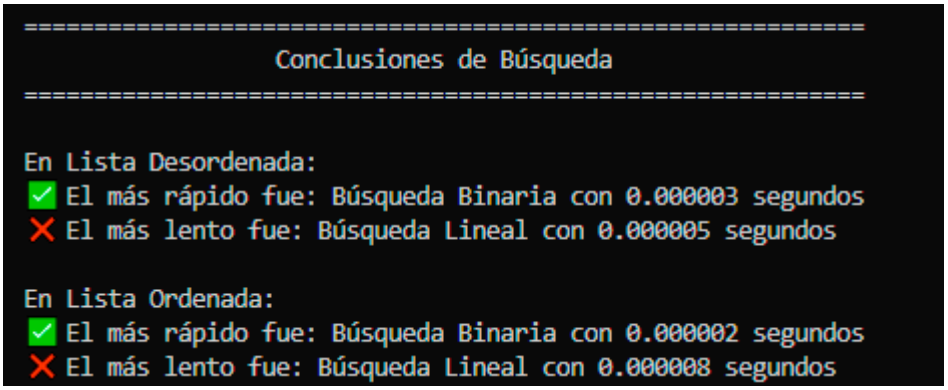
  Los resultados mostraron, como se dijo antes, que:

El algoritmo sorted() fue el más rápido tanto para ordenar la lista completa como para ordenar la lista acotada.

El algoritmo Selection Sort mostró un rendimiento más lento en listas completas, mientras que Quick Sort fue más lento en listas acotadas debido a la sobrecarga recursiva.



El algoritmo de búsqueda binaria fue el más rápido en ambas listas, pero el algoritmo de búsqueda lineal realizó correctamente su labor en ambos casos



- Comparación de rendimiento entre algoritmos:

  Algoritmo de ordenamiento:

  - Lista Completa (331 ciudades):

    Selection Sort fue 8868.75% más lento que sorted(), lo que significa que tomó 89.69 veces más tiempo para realizar la tarea de ordenar la lista completa.

  - Lista Top 10 Ciudades:

    Quick Sort fue 900% más lento que sorted(), lo que significa que tomó 10 veces más tiempo para ordenar la lista de las 10 primeras ciudades.

Algoritmo de búsqueda:

En lista desordenada:

- La búsqueda binaria fue ligeramente más rápida (0.000003 s), aunque este resultado es incorrecto en la práctica, ya que no debería usarse en listas desordenadas.
- La búsqueda lineal fue más lenta (0.000005 s), pero es la única válida en este caso.

En lista ordenada:

- La búsqueda binaria volvió a ser más rápida (0.000002 s), mostrando su eficiencia real cuando se cumplen las condiciones adecuadas.
- La búsqueda lineal tardó 0.000008 s, siendo claramente más lenta que la binaria.

- Enlace a repositorio en GitHub.

6. Conclusiones

El proyecto permitió comprender y comparar diferentes algoritmos de ordenamiento y búsqueda, destacando la importancia de elegir el método adecuado según el tamaño y el estado de los datos. Los algoritmos nativos de Python y la búsqueda binaria demostraron ser los más eficientes en listas grandes y ordenadas. Esta experiencia refuerza la aplicabilidad de estos conceptos en proyectos reales y abre la puerta a futuras mejoras como la incorporación de más algoritmos o herramientas visuales.

En resumen, el estudio demuestra que no existe un algoritmo universalmente óptimo. La elección adecuada depende del tamaño de los datos, si están ordenados o no, y del contexto de uso. Conocer las ventajas y limitaciones de cada método permite tomar decisiones más inteligentes en el desarrollo de soluciones eficientes.

7. Bibliografía

- W3Schools. (n.d.). *Python sort() method*. W3Schools. https://www.w3schools.com/python/ref_list_sort.asp
- W3Schools. (n.d.). *Python sorted() function*. W3Schools. https://www.w3schools.com/python/ref_func_sorted.asp
- SimpleMaps. (2024). Argentina Cities Database. Recuperado de https://simplemaps.com/data/ar-cities (Licencia: MIT)
- OpenAI. (2025). *ChatGPT (versión GPT-4)*. OpenAI. https://openai.com/chatgpt