Semana 14 – Sesión 1 (Sesión 27): Introducción a Ordenamiento y Búsqueda

Departamento de Física.

Corodinadora: C Loyola
Profesores C Femenías / F Bugini / D Basantes

Primer Semestre 2025

Universidad Andrés Bello
Departamento de Física y Astronomía

Universidad Andrés Bello®

python powered

Resumen – Semana 14, Sesión 1

Motivación

Ordenamiento básico

Búsqueda

Ejercicios en clase

Motivación

Motivación ∈ ¿Por qué ordenar y buscar?

- **Astronomía**: ordenar estrellas por brillo para ver cuál es la más tenue o la más brillante.
- **Física**: organizar mediciones de laboratorio (tiempos de caída, voltajes) antes de graficar.
- **Computación**: muchos algoritmos necesitan datos *ordenados* (p.ej. `binary search`).

Ordenamiento básico

Ordenamiento básico ∈ Bubble Sort paso a paso

```python
def bubble_sort(lista):
    n = len(lista)
    for i in range(n-1):              # pasadas
        for j in range(n-1-i):        # pares a comparar
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
```

- Compara **pares adyacentes** y “burbujea” el mayor al final.
- Complejidad: $\mathcal{O}(n^2)$.
- Fácil de codificar, ideal para entender intercambios. [1]

[1] Ver animaciones interactivas: HackerEarth :contentReference[oaicite:0]index=0

Ordenamiento básico ∈ Ejemplo – Ordenar distancias planetarias

```
dist_au = [1.52, 0.39, 5.20, 1.00, 30.07]   # Marte, Mercurio, Júpiter, Tierra, Neptuno
print("Original :", dist_au)
print("Ordenado  :", bubble_sort(dist_au))
```

Ordenamiento básico ∈ Ejemplo – Ordenar distancias planetarias

```
1  dist_au = [1.52, 0.39, 5.20, 1.00, 30.07]   # Marte, Mercurio,
   ↪  Júpiter, Tierra, Neptuno
2  print("Original :", dist_au)
3  print("Ordenado  :", bubble_sort(dist_au))
```

Resultado esperado: $[0.39, 1.00, 1.52, 5.20, 30.07]$

Búsqueda

Búsqueda ∈ Búsqueda lineal

```
1  def linear_search(lista, objetivo):
2      for i, x in enumerate(lista):
3          if x == objetivo:
4              return i
5      return -1   # no está
```

Búsqueda $\in$ Búsqueda binaria (en lista ordenada)

```python
def binary_search(lista_ordenada, objetivo):
    low, high = 0, len(lista_ordenada)-1
    while low <= high:
        mid = (low + high) // 2
        if lista_ordenada[mid] == objetivo:
            return mid
        elif objetivo < lista_ordenada[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

- Reduce el rango a la mitad cada vez $\longrightarrow \mathcal{O}(\log n)$.
- Sólo funciona si la lista está **ordenada**.
- Explicación amigable: Khan Academy.

Ejercicios en clase

Ejercicios en clase ∈ Ejercicio 1 – Preparar los datos de temperatura

Paso 1 – Obtener la lista T_C

```
import pandas as pd
import numpy as np

np.random.seed(42)
temps = np.random.normal(loc=22, scale=2,
↪  size=5_000).round(2).tolist()
```

Se crean 5000 *mediciones* con media 22°C.

Ejercicios en clase ∈ Ejercicio 1 – Ordenar y medir rendimiento

Tu burbuja

```
def bubble_sort(lst):
    n = len(lst)
    ....
```

Versión NumPy

```
import numpy as np
arr = np.array(temps)
arr_sorted = np.sort(arr)
```

Paso 2 – Comparar tiempos con %timeit (Jupyter/Colab)

```
%%timeit -n 3 -r 3
bubble_sort(temps.copy())    # tu implementación

%%timeit -n 3 -r 3
np.sort(arr)                 # NumPy optimizado
```

Las opciones `-n` y `-r` hacen que se ejecute el código 3 veces (n), luego repite eso 3 veces (r) para tener un total de 9 ejecuciones. Luego calcula la media y desviación estándar de esas 3 repeticiones. Después de ordenar: `minT = sorted_list[0]`, `maxT = sorted_list[-1]`,

Ejercicios en clase ∈ Magic Commands

El comando anterior `%timeit o %%timeit` se conocen como *magic commands*, que permiten extender las funcionalidades del entorno interactivo (como los *Jupyter Notebook*, o *Google Colab*). Ejemplos comunes de *magic commands*:

- `%timeit`: mide tiempos de ejecución de una línea de código.
- `%%timeit`: mide el tiempo de todo un bloque o celda de código.
- `%matplotlib inline`: habilita gráficos dentro del notebook.
- `%ls`, `%cd`, `%who`: comandos mágicos relacionados con el sistema de archivos o variables de entorno.

Ejercicios en clase ∈ Ejercicio 2 - Búsqueda Binaria

Simulamos 300 magnitudes con distribución normal

```
import numpy as np
mags = np.random.normal(8, 1.5, size=300).round(2)  # Simulación
```

Ordenamos la lista (requisito para búsqueda binaria)

```
mags_ordenadas = sorted(mags)
```

- ‘np.random.normal‘ genera datos aleatorios tipo Gaussianos.
- ‘round(2)‘ acorta los decimales.
- ‘sorted()‘ es obligatorio: la búsqueda binaria sólo funciona en listas ordenadas.

Ejercicios en clase ∈ Paso 2 – Definición de la función de búsqueda binaria

Función con contador de comparaciones

```python
def binary_search(lst, target):
    low, high, checks = 0, len(lst)-1, 0
    while low <= high:
        mid = (low + high) // 2
        checks += 1
        if lst[mid] == target:
            return mid, checks
        elif target < lst[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return -1, checks
```

- Divide la lista a la mitad en cada paso.
- ‘checks‘ cuenta cuántas comparaciones hicimos.

Ejercicios en clase ∈ Paso 3 – Ejecución y comparación

Buscar valor ingresado por el usuario

```
val = float(input("Ingresa magnitud a buscar: "))
pos, nchecks = binary_search(mags_ordenadas, val)
if pos != -1:
    print(f"Encontrado en índice {pos} tras {nchecks}
    ↪  comparaciones.")
else:
    print(f"No aparece (se hicieron {nchecks}
    ↪  comparaciones).")
```

- La búsqueda binaria es mucho más rápida que la búsqueda lineal.
- Búsqueda lineal: hasta 300 comparaciones.
- Búsqueda binaria: a lo más $\log_2(300) \approx 9$ comparaciones.

Ejercicios en clase ∈ Cómo evaluar tu búsqueda

- Añade un **contador global** en la función para registrar comparaciones.
- Realiza la misma búsqueda con un *recorrido lineal* y compara ambos contadores.
- Usa `%timeit` para medir el tiempo de la búsqueda lineal vs. binaria.

```
# búsqueda lineal (baseline)
def linear_search(lst, target):
    for checks, x in enumerate(lst, 1):
        if x == target:
            return checks
    return checks  # no encontrado

%timeit binary_search(mags_ordenadas, val)
%timeit linear_search(mags_ordenadas, val)
```

Ejercicios en clase ∈ Cierre

- Resumen: **bubble sort** (sencillo, lento) vs. `np.sort` (rápido).
- **Binary search** ahorra comparaciones en listas ordenadas.
- **Practica** `%timeit` con distintas longitudes de lista para ver la diferencia de escala.