Algoritmos Voraces (Greedy) 2

Introducción

En esta segunda clase abordaremos **problemas voraces más complejos**, donde la clave está en **tomar decisiones locales óptimas** para resolver problemas de optimización global.
Cada uno de estos ejercicios es un clásico en entrevistas técnicas y competiciones de programación.

---

1 Mínimo número de saltos para llegar al final

Dificultad: ★ ★

Descripción del problema:
Dado un arreglo `arr[]` donde cada elemento representa el **máximo número de pasos** que se puede avanzar desde esa posición, encuentra el **mínimo número de saltos** necesarios para llegar al último índice.
Si no es posible llegar, retorna `-1`.

```
def minJumps(arr):
    n = len(arr)
    if n <= 1:
        return 0
    if arr[0] == 0:
        return -1

    maxReach = arr[0]
    currReach = arr[0]
    jumps = 1

    for i in range(1, n):
        if i == n - 1:
            return jumps
        maxReach = max(maxReach, i + arr[i])
        if i == currReach:
            if maxReach <= i:
                return -1
            jumps += 1
            currReach = maxReach
    return -1
```

2 Conectar cuerdas con costo mínimo

Dificultad: ★ ★ ★

Descripción del problema:
Tienes `n` cuerdas de diferentes longitudes.
El costo de **conectar dos cuerdas** es igual a la **suma de sus longitudes**.
El objetivo es **conectar todas las cuerdas en una sola** con el **costo total mínimo posible**.

Ejemplo:

- Entrada: `[4, 3, 2, 6]`
- Salida: `29`

Explicación paso a paso:

1. Conectamos `2 + 3 = 5` → costo total = `5`
2. Conectamos `4 + 5 = 9` → costo total = `5 + 9 = 14`
3. Conectamos `6 + 9 = 15` → costo total = `14 + 15 = 29`

💡 **Idea del algoritmo (Greedy con Min-Heap):**
Para minimizar el costo, **siempre debemos conectar las dos cuerdas más cortas primero**.
Así, cada nueva conexión tiene el menor impacto posible en los siguientes pasos.

Esto se implementa fácilmente usando una **cola de prioridad (min-heap)**, que nos permite extraer y combinar las cuerdas más cortas de forma eficiente.

🤔 **Implementación en Python:**

```
import heapq

def minCostToConnectRopes(cuerdas):
    if not cuerdas:
        return 0

    # Crear un min-heap
    heapq.heapify(cuerdas)
    costo_total = 0

    # Conectar siempre las dos cuerdas más cortas
    while len(cuerdas) > 1:
        primera = heapq.heappop(cuerdas)
        segunda = heapq.heappop(cuerdas)
        costo = primera + segunda
        costo_total += costo
        heapq.heappush(cuerdas, costo)

    return costo_total


# Ejemplo de uso
if __name__ == "__main__":
    cuerdas = [4, 3, 2, 6]
    print(minCostToConnectRopes(cuerdas))  # 29
```

3 Selección de Actividades 🏃

Dificultad: ★ ★

Descripción del problema:
Dadas dos listas `start[]` y `finish[]` que representan los tiempos de inicio y fin de varias actividades, una persona solo puede realizar una actividad a la vez.
Queremos encontrar el **máximo número de actividades** que se pueden realizar sin superposición.

Ejemplo:

- Entrada:
  `start = [1, 3, 0, 5, 8, 5]`
  `finish = [2, 4, 6, 7, 9, 9]`
- Salida: `4`
- Explicación: Las actividades elegidas son las de índices `{0, 1, 3, 4}`.

Explicación de la solución (greedy):
Ordenamos las actividades por su **tiempo de finalización**.
Luego seleccionamos la primera y, sucesivamente, la siguiente que empiece **después de que termine la anterior**.
La elección local de la actividad que **acaba antes** permite maximizar el número total de actividades.

```
def activitySelection(start, finish):
    arr = list(zip(start, finish))
    arr.sort(key=lambda x: x[1])  # Ordenar por tiempo de finalización
    count = 1
    j = 0
    for i in range(1, len(arr)):
        if arr[i][0] > arr[j][1]:
            count += 1
            j = i
    return count
```

4 Fusionar intervalos superpuestos ⏱️

Dificultad: ★ ★

Descripción del problema:
Dada una lista de intervalos, donde cada intervalo representa un rango de tiempo `[inicio, fin]`, queremos **fusionar todos los intervalos que se solapan** para obtener un conjunto de intervalos **mutuamente exclusivos**.

Ejemplo 1:

- Entrada: `[[1, 3], [2, 4], [6, 8], [9, 10]]`
- Salida: `[[1, 4], [6, 8], [9, 10]]`

Explicación:
Los intervalos `[1, 3]` y `[2, 4]` se solapan, por lo tanto se fusionan en `[1, 4]`.

Los demás intervalos no se solapan y permanecen igual.

Ejemplo 2:

- Entrada: `[[7, 8], [1, 5], [2, 4], [4, 6]]`
- Salida: `[[1, 6], [7, 8]]`

Explicación:
Los intervalos `[1, 5]`, `[2, 4]` y `[4, 6]` se solapan entre sí, por lo que se fusionan en un solo intervalo `[1, 6]`.

Explicación de la solución (Greedy):
La estrategia voraz consiste en:

1. **Ordenar** los intervalos por su tiempo de inicio.
2. Recorrerlos secuencialmente, y para cada intervalo:
    - Si **no se solapa** con el anterior, se agrega a la lista de resultados.
    - Si **se solapa**, se fusionan extendiendo el tiempo final al máximo de ambos.

De esta forma, siempre mantenemos intervalos **no superpuestos**, construyendo la solución óptima de manera incremental.

Implementación en Python:

```python
def mergeOverlap(intervalos):

    if not intervalos:
        return []

    # Ordenar los intervalos por tiempo de inicio
    intervalos.sort()

    # 2Lista resultado
    resultado = [intervalos[0]]

    # Recorrer y fusionar si hay solapamiento
    for inicio, fin in intervalos[1:]:
        ultimo_inicio, ultimo_fin = resultado[-1]
        if inicio <= ultimo_fin:  # hay solapamiento
            resultado[-1][1] = max(ultimo_fin, fin)
        else:
            resultado.append([inicio, fin])

    return resultado


# Ejemplo de uso
if __name__ == "__main__":
```

```
    arr = [[7, 8], [1, 5], [2, 4], [4, 6]]
    print(mergeOverlap(arr))  # [[1, 6], [7, 8]]
```

5 Plataformas mínimas requeridas 🚆

Dificultad: ★ ★ ★

Descripción del problema:
Dadas las horas de **llegada** y **salida** de los trenes en una estación, queremos determinar **el número mínimo de plataformas necesarias** para que **ningún tren tenga que esperar**.

Ejemplo:

- Llegadas: `[9.00, 9.40, 9.50, 11.00, 15.00, 18.00]`
- Salidas: `[9.10, 12.00, 11.20, 11.30, 19.00, 20.00]`
- **Salida esperada:** `3`

Explicación:
Entre las 9.40 y 11.00 hay tres trenes simultáneamente en la estación, por lo que se requieren **3 plataformas**.

💡 Idea del algoritmo (Greedy):

1. Ordenar las listas de **llegadas** y **salidas**.
2. Recorrer ambas listas comparando la hora más próxima.
3. Si un tren llega antes de que otro salga → **necesitamos una nueva plataforma**.
4. Si un tren sale antes o al mismo tiempo que otro llega → **liberamos una plataforma**.
5. Llevar la cuenta del número máximo de plataformas usadas en cualquier momento.

🤓 Implementación en Python:

```
def minPlataformas(llegadas, salidas):
    n = len(llegadas)
    llegadas.sort()
    salidas.sort()

    plataformas = 1
    max_plataformas = 1
    i, j = 1, 0

    while i < n and j < n:
        # Si llega un tren antes de que otro salga
        if llegadas[i] <= salidas[j]:
            plataformas += 1
            i += 1
        else:
            # Un tren ha salido, liberamos una plataforma
            plataformas -= 1
            j += 1
```

```
            max_plataformas = max(max_plataformas, plataformas)

        return max_plataformas


    # Ejemplo de uso
    if __name__ == "__main__":
        llegadas = [9.00, 9.40, 9.50, 11.00, 15.00, 18.00]
        salidas = [9.10, 12.00, 11.20, 11.30, 19.00, 20.00]
        print(minPlataformas(llegadas, salidas))  # 3
```

Problemas: Clase 8 – Algoritmos Voraces (Greedy) 2