NOMBRE:

1 **3 puntos**

Un empresario alquila una sala para dar recitales. La oferta es que el precio es lo mismo sea cual sea la duración del concierto. Recibe `N` peticiones y quiere maximizar su beneficio. Su socio está convencido de que la mejor estrategia es elegir primero las solicitudes de menor duración.
Se pide que hagas *una función Python* que elija los conciertos utilizando tanto la estrategia que indica el socio como la óptima (que debes conocer) y que diga si ambas estrategias coinciden o no en *número de conciertos* para la instancia recibida como argumento (*es decir, la función devuelve un booleano*). Ejemplo de llamada:

```
dos_voraces([(1,3),(2,5),(4,7),(4,6)])
```

La tupla `(1,3)` indica que el concierto empieza en el instante `1` y termina en el instante `3`. Son compatibles dos conciertos tal que uno termina y el otro empieza en el mismo instante. Puedes asumir que existe una función auxiliar `solapan` que recibe 2 tuplas y nos devuelve `True` si solapan y `False` en otro caso. **También se pide:** ¿Cuál es el coste temporal?

Solución:

```
def duracion(s_t):
  s,t = s_t
  return t-s

def voraz_socio(solicitudes):
  alquilados = set()
  for concierto in sorted(solicitudes,key=duracion):
    if not any(solapan(concierto,otro) for otro in alquilados):
      alquilados.add(concierto)
  return alquilados

def finalizacion(s_t):
  s,t = s_t
  return t

def voraz_optimo(solicitudes):
  alquilados = set()
  tfin = min(s for (s,t) in solicitudes)
  for (s,t) in sorted(solicitudes,key=finalizacion):
    if tfin <= s:
      alquilados.add((s,t))
      tfin = t
  return alquilados

def dos_voraces(solicitudes):
  return len(voraz_optimo(solicitudes)) == len(voraz_socio(solicitudes))
```

El coste de este algoritmo está dominado por la función `voraz_socio`. Ambos algoritmos voraces requieren ordenar las actividades, lo cual tiene un coste $O(n \log n)$. Sin embargo, en `voraz_socio` es posible que cada vez que consideremos un concierto sea necesario compararlo con todos los anteriores, lo cual nos da un coste $O(n^2)$ mientras que `voraz_optimo` preserva el coste $O(n \log n)$.

2 1.5 puntos

Se ha resuelto, mediante programación dinámica, el cálculo del coste del camino más barato en el río Ebro. Hay `E` embarcaderos numerados de `0` a `E-1` y queremos ir del primero al último sabiendo que solamente podemos llegar a un embarcaderos desde los 2 anteriores. La función `c(i,j)` nos devuelve el coste de ir de un embarcadero `i` a otro `j`. Nos dan este código:

```
def rio_Ebro(E, c):
  C = [None]*E
  C[0] = 0
  C[1] = c(0,1)
  for i in range(2, E):
    C[i] = min(C[i-1]+c(i-1,i), C[i-2]+c(i-2,i))
  return C[E-1]
```

Y **se pide** que lo adaptes para recuperar la secuencia de embarcaderos utilizado el vector `C` (es decir, sin utilizar punteros hacia atrás).

Solución:

```
def rio_Ebro_camino(E, c):
  C = [None]*E
  C[0] = 0
  C[1] = c(0,1)
  for i in range(2, E):
    C[i] = min(C[i-1]+c(i-1,i), C[i-2]+c(i-2,i))
  last = E-1
  path = [last]
  while last != 0:
    if C[last] == C[last-1]+c(last-1,last):
      last -= 1
    else:
      last -= 2
    path.append(last)
  path.reverse()
  return C[E-1],path
```

3 5.5 puntos

Deseamos asfaltar los K kilómetros de carretera que separan dos ciudades. Tenemos N tipos de tramo con los que construir la carretera. Para cada tipo de tramos i ($1 \leq i \leq N$) nos indican:

- su longitud en kilómetros L_i,
- su coste en euros C_i,
- el número de tramos en stock S_i.

Los tramos deben ser utilizados enteros y nos hemos de limitar al stock. Queremos conseguir tramos que sumen exactamente los K kilómetros minimizando el coste total. **Se pide:**

1. Especificar formalmente el conjunto de soluciones factibles X, la función objetivo a minimizar f y la solución óptima buscada $\hat{x}$.
2. Una ecuación recursiva que resuelva el problema y la llamada inicial.
3. El algoritmo iterativo asociado a la ecuación recursiva anterior para calcular *el menor coste* (no hace falta determinar el número de tramos necesarios de cada tipo).
4. El coste temporal y espacial del algoritmo iterativo.

Solución:

1. Especificar formalmente el conjunto de soluciones factibles X, la función objetivo a minimizar f y la solución óptima buscada $\hat{x}$.

   Podemos representar las soluciones mediante una tupla de longitud N donde cada elemento indica el número de tramos de cada tipo:

   $$X = \left\{ (x_1, \ldots, x_N) \in \mathbb{N}^N | 0 \leq x_t \leq S_t, 1 \leq t \leq N, \sum_{t=1}^{N} x_t L_t = K \right\}$$

   La función objetivo es $f((x_1, \ldots, x_N)) = \sum_{t=1}^{N} x_t C_t$, mientras que la solución óptima buscada es $\hat{x} = \text{argmin}_{x \in X} f(x)$.

2. Una ecuación recursiva que resuelva el problema y la llamada inicial.

   $$M(t,k) = \begin{cases} 0, & \text{si } t = 0 \text{ y } k = 0 \\ \infty, & \text{si } t = 0 \text{ y } k > 0 \\ \min\limits_{0 \leq x_t \leq \min(S_t, \lfloor k/L_t \rfloor)} M(t-1, k - x_t L_t) + x_t C_t, & \text{en otro caso} \end{cases}$$

   La llamada inicial sería $M(N, K)$. Este problema es virtualmente idéntico al cambio de monedas con limitación del número de monedas, excepto que el coste de usar cada moneda (aquí un tramo) varía según su tipo.

3. El algoritmo iterativo asociado a la ecuación recursiva anterior para calcular *el menor coste* (no hace falta determinar la secuencia de tramos a utilizar).

```
def tramos(K,L,C,S):
  # K es un entero
  # L, C, S son listas de longitud N
  N, inf = len(L), 2**31
  M = { (0,k):inf for k in range(K+1) }
  M[0,0] = 0 # caso base
  for t in range(N): # tipo de tramo
    for k in range(K+1): # distancia
      M[t,k] = min(M[t-1,k-x*L[i]]+x*C[i] for x in range( min(k//L[i],S[i]) +1))
  return M[N,K]
```

4. El coste temporal y espacial del algoritmo iterativo.

   El coste espacial de este algoritmo es $O(NK)$, mientras que el coste temporal es $O(NK^2)$ que corresponden a los costes vistos en teoría para el problema del cambio de monedas con limitación del número de monedas puesto que hacer distinto el coste de cada tramo no varía la complejidad espacial o temporal del algoritmo.