1. TEORÍA (Grupo B) — 1,5 puntos

Escribe el algoritmo iterativo de Programación Dinámica (en Python 3 o en seudocódigo) que calcula la siguiente recursión eficientemente, considerando que $b(x)$ se calcula en tiempo constante. La llamada inicial sería $f(X)$. Calcula el coste espacial y temporal justificando tus respuestas.

$$f(x) = \begin{cases} b(x) & \text{si } x \leq 1 \\ b(x) + \text{máx}_{1 \leq k < x} f(k) & \text{si } x > 1 \end{cases}$$

Solución:
Una posible solución donde se asume que la función `b` está en el ámbito (*scope*) de la función (otra opción sería pasar `b` como parámetro):

```
def teo1B(x):
    f = [b(0), b(1)]
    for i in range(2, x+1):
        f.append(b(i) + max(f[1:]))
    return f[x]
```

El coste espacial es debido a f, $O(x)$ (lineal), mientras que el coste temporal es $O(x^2)$ puesto que corresponde básicamente a $\sum_{i=1}^{x} i \approx \frac{x^2}{2}$, esto es, el coste del bucle `for` y la maximización.

Es posible mejorar tanto el coste espacial como el temporal si observamos que $\text{máx}_{1 \leq k < x} f(k) = \text{máx}(\text{máx}_{1 \leq k < x-1} f(k), f(x-1))$, con lo que podemos guardarnos únicamente el mayor valor de $f(k)$ (desde $k = 1$) hasta el momento:

```
def teo1B(x):
    if x<1:
        return b(x)
    # x>=1
    fx = b(1)
    maxf = fx # max f(k) para 1<=k<x
    for k in range(2, x+1):
        fx = b(k) + maxf
        maxf = max(maxf, fx) # para la proxima iteracion
    return fx
```

Con esta nueva versión no sólo tenemos una version con reducción del coste espacial (coste espacial $O(1)$) sino también con un menor coste temporal (ahora sería $O(x)$ o lineal).

2. PRÁCTICAS-PRUEBA OBJETIVA (Grupo B) 1,5 puntos

Escribe en la siguiente **matriz los valores** que se obtendrían al ejecutar el algoritmo de Programación Dinámica que resuelve el problema de la Mochila Sin Fraccionamiento para una mochila de capacidad $W = 12$ y con 5 objetos, de pesos $(3, 7, 4, 2, 5)$ y valores $(1, 4, 6, 4, 2)$. Podéis dejar en "blanco" las casillas que no son "alcanzables", es decir, que no tienen valor generado por los pesos y valores de los objetos. Indica asimismo el **valor máximo** que proporcionaría el algoritmo.
Puedes dibujar en papel una cuadrícula como la que te mostramos a continuación (no es preciso un perfecto paralelismo en las líneas) y escribe en ella la respuesta.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2					4	4
3		1	1	1	1	1
4				6	6	6
5					5	5
6					10	10
7			4	7	7	7
8						3
9					11	11
10			5	5	5	7
11				10	10	**12**
12					9	9

El resultado es el máximo de la última columna, es decir, el valor 12. Para obtener la solución óptima, esto es, qué objetos son los que generan este máximo beneficio, se deberían usar "backpointers" durante la resolución del problema o bien post-procesar la matriz de resultados intermedios. Para esta instancia, los objetos que forman parte de la solución óptima son los tres últimos.

3. TEORÍA (Grupo B) — 4,5 puntos

Una ONG tiene que llevar M toneladas de comida a P poblaciones distintas, y para ello tiene preparadas unas cantidades $c(i)$ de toneladas de comida, para cada una de las poblaciones i, $1 \leq i \leq P$. La ONG recorre las poblaciones secuencialmente repartiendo la comida, pero no es necesario asignar toda la cantidad reservada para ellas. Las cantidades no asignadas pueden guardarse y asignarse a los siguientes poblados, pero necesariamente se ha de distribuir toda la comida entre los P poblados. El beneficio de asignar una cantidad m a una población i viene dado por la función $b(m, i)$. Diseña un algoritmo de Programación Dinámica que proporcione el máximo beneficio que se puede conseguir. Para ello:

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

b) Calcula el máximo beneficio que se puede conseguir usando una ecuación recursiva de Programación Dinámica y la expresión que devuelve el valor pedido (es decir, la(s) llamada(s) inicial(es)).

c) Escribe el algoritmo iterativo de Programación Dinámica (en Python 3 o en seudocódigo) que resuelva eficientemente la ecuación anterior, y su coste espacial y temporal, justificando tus respuestas. Asume que los costes de las funciones b y c son constantes.

   Puedes utilizar el siguiente esquema a completar:

```
def ayudaONG(M, P, c, b):
    """
    M es el total de comida
    P es el número de poblaciones
    c es una función que puedes llamar c(i)
    b es una función que puedes llamar b(m,i)
    La función ayudaONG devuelve el máximo beneficio total
    """
    # completar aquí...
```

d) ¿Podría reducirse el coste espacial del algoritmo iterativo anterior? Justifica tu respuesta.

Solución:

a) Se puede formalizar de la siguiente manera:

$$X = \{(m_1, m_2, \ldots, m_P) \mid 0 \leq m_i \leq c(i) + \sum_{j=1}^{i-1}(c(j) - m_j), \sum_{j=1}^{P} m_j = M\}$$

$$f((m_1, m_2, \ldots, m_P)) = \sum_{i=1}^{P} b(m_i, i)$$

$$\hat{x} = \operatorname*{argmax}_{x \in X} f(x)$$

   Fíjate que la comida que se puede asignar a una población i puede valer desde 0 (no le asigno nada) al máximo preparado de antemano ($c(i)$) al que añado los excedentes de las asignaciones previas: $c(i) + \sum_{j=1}^{i-1}(c(j) - m_j)$. La otra restricción para que una asignación sea factible es que finalmente se debe distribuir toda la comida: $\sum_{j=1}^{P} m_j = M$.

b) Calcula el máximo beneficio que se puede conseguir usando una ecuación recursiva de Programación Dinámica y la expresión que devuelve el valor pedido (es decir, la(s) llamada(s) inicial(es)).

Sea $F(m, i)$ el máximo beneficio que se puede obtener destinando un total de m toneladas de comida en las i primeras poblaciones:

$$F(m,i) = \begin{cases} b(m,1) & \text{si } i = 1,\ 0 \leq m \leq c(1) \\ -\infty & \text{si } i = 1,\ m > c(1) \\ \text{máx}_{\forall\ 0 \leq m' \leq \text{mín}(m, \sum_{j=1}^{i-1} c(j))} \{b(m-m', i) + F(m', i-1)\} & \text{si } i > 1,\ 0 \leq m \leq \sum_{j=1}^{i} c(j) \\ -\infty & \text{si } i > 1, m > \sum_{j=1}^{i} c(j) \end{cases}$$

El caso general está descrito en términos de la cantidad m' que reservamos para las etapas anteriores (i.e. $F(m', i-1)$), que ha de ser un valor menor o igual a m y no superior a lo que hay disponible en esa etapa anterior ($\sum_{j=1}^{i-1} c(j)$).

La llamada inicial sería $F(M, P)$.

c) La versión iterativa que devuelve el beneficio máximo (sin recuperar la asignación a cada población):

```
def ayudaONG(M, P, c, b):
    """
    M es el total de comida
    P es el número de poblaciones
    c es una función que puedes llamar c(i)
    b es una función que puedes llamar b(m,i)
    La función ayudaONG devuelve el máximo beneficio total
    """
    F = {}
    maxtoneladas = c(1)
    for m in range(maxtoneladas+1):
        F[m,1] = b(m,1)
    for i in range(2,P+1): # avanzamos por el resto de poblaciones
        antiguamaxtoneladas = maxtoneladas
        maxtoneladas += c(i)
        for m in range(maxtoneladas+1):
            F[m,i] = max(b(m-mprime,i) + F[mprime,i-1]
                         for mprime in range(min(m,antiguamaxtoneladas+1)))
    # en este punto maxtoneladas == M
    return F[M,P]
```

El coste espacial del algoritmo es $O(MP)$, debido a la matriz de resultados intermedios. El coste temporal del algoritmo es $\sum_{n=1}^{P} M^2$, es decir $O(PM^2)$, debido a los tres bucles anidados: bucle `for` que recorre las poblaciones, con un coste $O(P)$, bucle `for` que recorre las toneladas, con un coste $O(M)$, y la maximización interior con también coste $O(M)$.

d) Sí es posible reducir el coste espacial de $O(MP)$ a $O(M)$ porque cada etapa i depende únicamente de la etapa anterior $i-1$ y con un vector columna de tamaño $O(M)$ es suficiente.

4. TEORÍA (Grupo B) **2,5 puntos**

Dada una secuencia S de n números, encuentra una **subsecuencia de valores estrictamente decrecientes que sumen un valor** V. Una subsecuencia de una secuencia es una nueva secuencia tal que todos sus números aparecen en la secuencia original y en el mismo orden relativo. Por ejemplo, (30, 3, 10) es una subsecuencia de (20, 30, 3, 20, 39, 3, 1, 10, 2): tanto el 30 como el 3 y el 10 aparecen en la segunda secuencia y lo hacen en el mismo orden relativo: el 30 antes que el 3 y el 3 antes que el 10. Buscamos además que la subsecuencia sea de valores estrictamente decrecientes y que sumen un cierto valor. He aquí una instancia concreta del problema: "Dada la secuencia S = (20, 30, 3, 20, 39, 3, 1, 10, 2), encuentra una subsecuencia de valores estrictamente decrecientes que sume V = 24." La respuesta es la subsecuencia (20, 3, 1).
Queremos diseñar un algoritmo de Búsqueda con Retroceso para resolver este problema basado en la expresión de una solución como vector con n booleanos: un valor "cierto" (o 1) quiere decir que el número que ocupa esa misma posición en la secuencia forma parte de la subsecuencia, y un valor "falso" (o 0) significa que no forma parte. Por ejemplo, dada la secuencia (20, 30, 3, 20, 39, 3, 1, 10, 2), la subsecuencia (30, 3, 10) se describe con el vector (0, 1, 1, 0, 0, 0, 0, 1, 0).
Debes responder a las siguientes cuestiones:

a) ¿Qué es un estado? ¿Qué aspecto tiene? (Pon algún ejemplo tomando como base la instancia presentada en el enunciado.)

b) ¿Cómo ramificas un estado? (Pon algún ejemplo, de nuevo, tomando como base la instancia del enunciado.) ¿Cuántos hijos puede tener en el mejor caso?

c) ¿Qué comprobaciones harías sobre un estado para determinar que es prometedor?

d) ¿Y para determinar que es completo?

e) ¿Y para determinar que, siendo completo, es factible?

f) Por último, escribe un algoritmo (en Python 3 o en seudocódigo) que resuelva el problema. Puedes utilizar el siguiente esquema a completar:

```
def estrDecrecientesSumanV(seq, V):
    """
    seq es una lista con la secuencia
    V el valor a sumar
    devuelve una lista de 0s y 1s
    """
```

Por ejemplo, la llamada:

```
estrDecrecientesSumanV([20,30,3,20,39,3,1,10,2], 24)
```

devolvería:

```
[1, 0, 1, 0, 0, 0, 1, 0, 0]
```

Solución:

a) Un estado es un conjunto de soluciones, se puede representar como una solución incompleta, por ejemplo, el estado (0,0,1,?) representa todas las soluciones que lo contienen como prefijo. Podemos representar ese estado con el vector (0,0,1).

b) Dado un estado, la ramificación tomará una decisión sobre qué hacer con el valor siguiente de la secuencia: solo hay dos opciones, incluirlo en la secuencia (valor 1) o no incluirlo (valor 0). En el mejor caso tiene 2 hijos. Un ejemplo: $branch(1, 0, ?) = \{(1, 0, 1, ?), (0, 1, 0, ?)\}$. Pero puede suceder que solo tenga un hijo: si el siguiente valor NO es menor que el elemento anterior, solo puede tener un hijo: $branch(1, 0, 1, ?) = \{((1, 0, 1, 0, ?)\}$.

c) Si un estado $(x,x_2,\ldots,x_i,?)$ es prometedor, su hijo $(x,x_2,,\ldots,x_i,0,?)$ también lo será, no hacer falta comprobar nada. Pero para que el hijo $(x,x_2,,\ldots,x_i,1,?)$ sea prometedor se deben cumplir las dos condiciones: que su valor asociado sea menor que el anterior anterior considerado y que no se sobrepase el valor V (o bien poner estas condiciones en la ramificación).

d) Para decidir que un estado es completo se debe cumplir que se hayan tomado las N decisiones o que se haya alcanzado el valor V.

e) Un estado completo solo será factible si la suma es V.

f) Una posible solución Python:

```
def estrDecrecientesSumanV(seq, V):
    """
    seq es una lista con la secuencia
    V el valor a sumar
    devuelve una lista de 0s y 1s
    """
    N = len(seq)
    solucion = [0]*N
    def backtracking(longsol, pesoAcum, pesoUltimo):
        if pesoAcum == V:
            return solucion
        elif longsol < N: # si no entra aquí devolverá None
            # ramificar el 1 primero, aunque no siempre:
            if (pesoUltimo is None or
                (pesoUltimo > seq[longsol] and
                 V <= pesoAcum + sum(seq[i] for i in range(longsol,N)
                                     if pesoUltimo > seq[i]))):
                solucion[longsol] = 1
                resul = backtracking(longsol+1,
                                     pesoAcum+seq[longsol],
                                     seq[longsol])
                if resul is not None:
                    return resul
                solucion[longsol] = 0
            # ramificar el 0 siempre:
            resul = backtracking(longsol+1, pesoAcum, pesoUltimo)
            if resul is not None:
                return resul

    return backtracking(0, 0, None)
```

Fijaos que en el programa anterior hemos introducido varias mejoras con respecto a una solución más directa:

- Se usa una variable global para construir la solución en lugar de pasarla como argumento al método recursivo (solo se pasa la posición que estamos considerando).
- Se ramifica en primer lugar el estado que considera añadir un nuevo elemento (si ello es posible) en lugar de no añadirlo.
- La variable `pesoAcum` nos evita tener que sumar el valor acumulado en un estado cada vez que consideramos un nuevo estado: solo es necesario comprobar si ese valor acumulado más el nuevo elemento no sobrepasan el valor V (en caso de añadir un nuevo elemento).
- Se usa una condición de "estado prometedor" que no es estrictamente necesaria: el peso acumulado más la suma de todos los elementos que quedan por considerar y que sean menor estricto que el último de la subsecuencia debe alcanzar el valor V (es decir, debe ser $\geq V$).

1. PRÁCTICAS (Grupo B) — 10 puntos

Completa la función `permutaMAYOR` que recibe los valores `N` y `K` y que, utilizando al técnica de *backtracking* o búsqueda con retroceso, imprime por pantalla aquellas permutaciones de los números del `0` al `N-1` que cumplan lo siguiente:

> Un valor `n` sólo puede situarse en una posición (índice de la lista Python) `i` si `abs(n-i)>k`.

Nota: cuando se alcanza una solución basta con imprimirla usando `print`, no hace falta utilizar `yield`. La función tampoco debe devolver nada, simplemente limitarse a imprimir las soluciones por pantalla.
Por ejemplo, la llamada `permutaMAYOR(5,1)` produce la siguiente salida por pantalla:

```
[2, 3, 4, 0, 1]
[2, 3, 4, 1, 0]
[3, 4, 0, 1, 2]
[4, 3, 0, 1, 2]
```

Como sugerencia se propone el siguiente esquema a completar:

```
def permutaMAYOR(N, K):
    seq = [None] * N # una lista de longitud N
    def backtracking(i): # longitud de la solución parcial
        # completar
    # completar
```

Solución: Una posible solución:

```
def permutaMAYOR(N, K):
    seq = [None] * N # una lista de longitud N
    def backtracking(i): # longitud de la solución parcial
        if i >= N:
            print(seq)
        else:
            for n in range(N):
                if n not in seq[:i] and abs(n-i)>k:
                    seq[i] = n
                    backtracking(i+1)
    backtracking(0)
```