

Tenemos N varillas distintas de longitudes naturales l_1, l_2, \dots, l_N , con $l_i \in \mathbf{N}^{>0}$. Cada varilla tiene asociado un precio $p_i \in \mathbf{R}^{>0}$. Las varillas no se pueden cortar. Se desea soldar algunas de ellas para obtener una única varilla de longitud total L de forma que se **minimice el precio global**.

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 calcule el precio global mínimo (o infinito si no se puede obtener la varilla deseada).
3. El algoritmo iterativo (preferiblemente en Python3) asociado a la ecuación recursiva anterior.
4. Calcular el coste temporal y espacial del algoritmo iterativo. Justifica brevemente tus respuestas.
5. ¿Se puede reducir el coste espacial o temporal de tu algoritmo? Justifica brevemente tus respuestas.

Este problema es asimilable al problema de la mochila discreta, con la única diferencia de que debemos alcanzar exactamente la longitud total L eligiendo un subconjunto de las N varillas (esto es, es “necesario llenar completamente la mochila”). Así, la formalización del problema es:

$$X = \left\{ (x_1, \dots, x_N) \in \{0, 1\}^N \mid \sum_{i=1}^N x_i l_i = L \right\}$$

La función objetivo es $f((x_1, \dots, x_N)) = \sum_{i=1}^N x_i p_i$, mientras que la solución óptima buscada es $\hat{x} = \operatorname{argmin}_{x \in X} f(x)$.

La llamada $M(N, L)$ a la siguiente ecuación recursiva resuelve el problema:

$$M(i, l) = \begin{cases} 0, & \text{si } i = 0 \text{ y } l = 0 \\ +\infty, & \text{si } i = 0 \text{ y } l > 0 \\ \min(M(i-1, l), M(i-1, l-l_i) + p_i), & \text{si } i > 0 \text{ y } l \geq l_i \\ M(i-1, l), & \text{si } i > 0 \text{ y } l < l_i \end{cases}$$

El algoritmo iterativo asociado a la ecuación recursiva anterior para calcular *el menor precio*:

```
def menorprecio(L, l, p):
    # L es un entero, la longitud a obtener
    # l, p son listas de longitud N
    N, inf, M = len(l), 2**31, {}
    M[0,0] = 0
    for l in range(1, L+1):
        M[0,l] = inf
    for i in range(1, N+1):
        for l in range(0, l[i]):
            M[i,l] = M[i-1,l]
        for l in range(l[i]+1, L+1):
            M[i,l] = min(M[i-1,l], M[i-1,l-l[i]]+p[i])
    return M[N,L]
```

El coste espacial de este algoritmo es $O(NL)$, como se observa con los bucles utilizando i y l donde se rellena $M[i, l]$. El coste temporal es asimismo $O(NL)$, que corresponde a los bucles anteriores. El coste espacial se puede reducir utilizando únicamente dos vectores columna de tamaño L .

Nos interesa calcular la subsecuencia **estrictamente creciente** más larga de una secuencia de N números. Por ejemplo, dada la secuencia $x = 4, 1, 1, 2, 6, 3, 5, 8$, la subsecuencia creciente más larga es $1, 2, 3, 5, 8$, de talla 5. La siguiente ecuación recursiva permite resolver este problema mediante Programación Dinámica:

$$f(i) = \begin{cases} 0, & \text{si } i = 0 \\ 1 + \max_{\substack{0 \leq j < i \\ x_j < x_i}} f(j), & \text{si } i > 0 \end{cases}$$

Donde, para evitar problemas con el máximo cuando no hay ningún elemento anterior con valor inferior, se asume que $x_0 = -\infty$.

Se pide:

1. Escribe un algoritmo iterativo asociado a la ecuación recursiva anterior. El algoritmo debe recibir como entrada una lista/vector python como la siguiente que corresponde al ejemplo dado, $[-2 * 31, 4, 1, 1, 2, 6, 3, 5, 8]$, y debe obtener la talla de la subsecuencia estrictamente creciente más larga.
2. Calcula el coste temporal y espacial del algoritmo iterativo. Justifica brevemente tus respuestas.

Solución: Se pide una transformación recursiva-iterativa de programación dinámica. Será necesario un almacén de resultados intermedios de tamaño N y el orden vendrá dado por la propia secuencia, de 0 a N . El coste espacial vendrá determinado por el tamaño del almacén, $O(N)$, y el coste temporal es $O(N^2)$, ya que hay resolver N problemas y para cada uno, maximizar por todos los elementos anteriores, del orden de $O(N)$.

```
def subseq(x):
    # x es la lista de longitud N con la secuencia de números
    # F es el almacén de resultados intermedios, de talla 0..N
    N, F = len(x), {}
    F[0] = 0
    for i in range(1, N+1):
        F[i] = max(F[j]+1 for j in range(i) if x[j]<x[i])
    return F[N]
```

Dados N puntos distintos de la recta real x_1, \dots, x_N ya ordenados: $x_1 < x_2 < \dots < x_N$, queremos agruparlos en **conjuntos de puntos consecutivos** con al menos un punto por grupo: C_1, C_2, \dots, C_m . Al haber al menos un punto por grupo, es obvio que $m \leq N$. Por ejemplo, para 8 puntos (x_1, x_2, \dots, x_8) , dos posibles agrupaciones serían:

- a) $C_1 = \{x_1, x_2, x_3\}, C_2 = \{x_4\}, C_3 = \{x_5, x_6\}, C_4 = \{x_7, x_8\}$
- b) $C_1 = \{x_1, x_2\}, C_2 = \{x_3, x_4, x_5, x_6, x_7\}, C_3 = \{x_8\}$

La siguiente función mide la bondad de un agrupamiento C_1, C_2, \dots, C_m :

$$\sum_{k=1}^m \alpha(s_k, e_k)$$

donde $\alpha(a, b)$ con $1 \leq a \leq b \leq N$ mide lo bueno que es agrupar los puntos x_a, x_{a+1}, \dots, x_b en un mismo grupo, y donde s_k y e_k son los extremos (s de *start*, e de *end*) del grupo C_k . Por ejemplo, la bondad calculada para los ejemplos anteriores sería, respectivamente:

- a) $\alpha(1, 3) + \alpha(4, 4) + \alpha(5, 6) + \alpha(7, 8)$

b) $\alpha(1, 2) + \alpha(3, 7) + \alpha(8, 8)$

El siguiente código calcula el máximo valor de bondad de la mejor agrupación utilizando un diccionario para almacenar los resultados intermedios:

```
def grupos(N,alpha):
    F = { 0:0 }
    for k in range(1,N+1):
        F[k] = max(F[j]+alpha(j+1,k) for j in range(k))
    return F[N]
```

Se pide:

1. Realiza los cambios necesarios que consideres para que la función devuelva adicionalmente al valor de bondad máximo, la agrupación correspondiente.
2. Calcula el coste temporal y espacial del algoritmo iterativo proporcionado, así como el diseñado por tí, sabiendo que $\alpha(a, b)$ es $\theta(1)$. Justifica brevemente tus respuestas.

Es necesario crear un almacén de “backpointers” de idéntico tamaño a F , e ir almacenando el mejor predecesor para poder obtener la mejor agrupación con un postproceso. El coste este nuevo algoritmo no difiere del algoritmo original, siendo $O(N)$ el coste espacial debido a los diccionarios F y B ; y $O(N^2)$ el coste temporal por el doble bucle k y j . El postproceso sobre B tiene un coste $O(N)$.

```
def grupos(N,alpha):
    F = { 0:0 }
    B = { 0:None }
    for k in range(1,N+1):
        F[k], B[k] = max(F[j]+alpha(j+1,k), j for j in range(k))
    agrupacion = []
    k = N
    while B[k] != None:
        k = B[k]
        agrupacion.append (k+1)
    agrupacion.reverse()
    return F[N], agrupacion
```

4

1 punto

Realiza una traza de un algoritmo de programación dinámica (versión iterativa) del problema del cambio de una cierta cantidad de dinero Q con el menor número posible de monedas de N valores (suponiendo un número ilimitado de monedas de cada tipo) para la siguiente instancia: $Q = 17$ y $N = 6$ tipos de monedas de valores $\{1, 2, 5, 10, 20, 50\}$. ¿Coincide en este caso la solución óptima con la solución devuelta por un algoritmo voraz que resuelva la misma instancia?

Rellena el contenido del almacén de resultados intermedios ($F[i]$ contendrá el menor número de monedas para devolver la cantidad i) e indica qué devuelve el algoritmo:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
F	1	1	2	2	1	2	2	3	3	1	2	2	3	3	2	3	3

El algoritmo devuelve el contenido de $F[Q] = F[17]$ que contiene el valor 3. Para esta instancia, sí que coincide con el algoritmo voraz (ir eligiendo la moneda de mayor valor mientras se pueda: un billete de 10 euros, 1 moneda de 5 euros y 1 moneda de 2 euros).