Enunciado

Una biblioteca tiene interés en conseguir n artículos académicos que fueron publicados en m volúmenes de un mismo periódico. Cada artículo se publicó en un único volumen, mientras que cada volumen contiene varios artículos. La biblioteca puede adquirir cada artículo en forma individual, o puede comprar el volumen en que salió publicado dicho artículo. En el segundo caso, la biblioteca adquiere todos los artículos publicados en el volumen. Como hay un presupuesto acotado, la biblioteca no puede comprar todos los artículos. Por este motivo, sus bibliotecaries decidieron priorizar los artículos, asignándoles un valor. El objetivo de la biblioteca es maximizar el valor total de los artículos adquiridos (i.e., la suma de los valores individuales) sin superar su presupuesto.

Para cada artículo $1 \leq i \leq n$ se conoce su valor $v_i \in \mathbb{N}$, su volumen e_i ($1 \leq e_i \leq m$) y su costo $c_i \in \mathbb{N}$ si se compra en forma individual. Para cada volumen $1 \leq j \leq m$ se conoce su costo $p_j \in \mathbb{N}$. Para simplicar el ejercicio, suponemos que $e_i \leq i$ (y por ende $m \leq n$) y que $e_1 \leq e_2 \leq \cdots \leq e_n$. Es decir, los artículos que fueron publicados en el j-ésimo volumen son consecutivos. Ejemplo: si el presupuesto es 10, los artículos valen y cuestan $(4, 5, 3, 2, 4)$, los volúmenes son $(1, 1, 2, 3, 3)$ y cuestan $(7, 4, 4)$, entonces el máximo valor posible es 12 y se consigue adquiriendo el primer volumen y el tercer artículo.

- Definir en forma recursiva la función $B : \{1, \ldots, n\} \times \mathbb{N} \to \mathbb{N}$ tal que $B(i, q)$ denota el máximo valor total que puede obtener la biblioteca cuando su presupuesto es q, si compra únicamente artículos menores o iguales a i y volúmenes que no tienen artículos mayores a i.
- Demostrar que B tiene la propiedad de superposición de subproblemas.
- Definir un algoritmo top-down para calcular $B(i, q)$ cuando se obtiene el input indicado previamente, indicando claramente las estructuras de datos utilizadas y la complejidad resultante.
- Escribir el (pseudo-)código del algoritmo top-down resultante.

Análisis

El máximo valor que puedo obtener, si tengo q pesos, y puedo comprar sólo artículos anteriores o iguales al i-ésimo, y volúmenes que no tengan artículos posteriores al i-ésimo, es, en orden:

- Si $q < 0$, es $-\infty$, porque no hay manera de llegar a esta condición, y el máximo del conjunto vacío es $-\infty$.
- Si $i < 1$, entonces no hay artículos que comprar, menos aún volúmenes. Luego, el máximo valor que puedo obtener es comprando todos los cero posibles libros disponibles, pagando $0 \leq q$, y obteniendo un valor de 0.
- Si el i-ésimo artículo es el último de su volúmen, sé que todos los otros artículos de este volúmen tienen índices menores que i. Por lo tanto, según la definición de B, puedo:
  - Comprar el volumen que contiene a i. Si el volumen que contiene al artículo i es el j, entonces esto me cuesta p_j] pesos, y me otorga un valor w_j, que defino como la suma de los valores de todos los artículos del volumen j.
    Habiendo comprado el volumen entero, sólo voy a poder comprar artículos cuyo índice sea menor al primer índice del volumen j. Como i era el último artículo del volumen j, entonces llamando l_j al número de artículos en el volumen j, además de comprar este volumen, puedo comprar artículos que tengan índice menor o igual a $i - l_j$.
    Esto es $B(i - l_j, q - p_j) + w_j$.
  - Comprar el artículo i individualmente, y además comprar artículos que tengan índice menor o igual a $i - 1$. Esto me cuesta c_i pesos, y me da un valor de v_i.
    Esto es $B(i - 1, q - c_i) + v_i$.
  - No comprar el artículo i ni el volumen al que pertenece, y comprar artículos con índice menor o igual a $i - 1$. Esto me da un valor de 0, y me cuesta 0.
    Esto es $B(i - 1, q)$.

  Quiero la opción que me de un mayor valor. Luego esto es:
  $\max\{B(i - 1, q), B(i - 1, q - c_i) + v_i, B(i - l_j, q - p_j) + w_j)\}$.

- De otra forma, el i-ésimo artículo no es el último de su volumen. Luego, por la definición de B, no puedo comprar el volumen al que pertenece, pues el volumen contiene al menos un artículo posterior al i-ésimo, y luego ese artículo contiene un índice mayor a i. Luego, este caso es análogo al anterior, sólo que no puedo comprar el volumen. Sigo pudiendo comprar el artículo individualmente, y no-comprarlo.

  Esto es, entonces: $\max\{(B(i-1,q), B(i-1,q-c_i)+v_i)\}$.

Luego, nuestra función recursiva es:

$$B(i,q) = \begin{cases} -\infty & \text{si } q < 0 \\ 0 & \text{si } i < 1 \\ \max\left\{\begin{array}{l} B(i-1,q), \\ B(i-1,q-c_i)+v_i, \\ B(i-l_{e_i}, q-p_{e_i})+w_{e_i} \end{array}\right\} & \text{si } (i=n) \text{ o } (e_i \neq e_{i+1}) \\ \max\left\{\begin{array}{l} B(i-1,q), \\ B(i-1,q-c_i)+v_i \end{array}\right\} & \text{de otra forma} \end{cases}$$

donde $l_k = |\{i \mid 1 \leq i \leq n, e_i = k\}|$ es el número de artículos en el k-ésimo volumen, y $w_k = \sum_{\substack{1 \leq i \leq n, \\ e_i = k}} v_i$ es la suma de los valores de todos los artículos en el k-ésimo volumen.

Semánticamente, los colores significan:

- No comprar el artículo.
- Comprar el artículo i individualmente.
- Comprar el volumen al que pertenece el artículo i.
- El artículo i es el último de su volumen.

Para resolver el problema, llamamos a $B(n,q)$. Esto nos devuelve el máximo valor posible, si podemos comprar cualquiera de los n artículos y/o sus volúmenes, y contamos con q pesos.

Implementación

Una implementación recursiva trivial de esto, usando índices que empiezan en 0 en vez de 1 para hacerlo más idiomático en Python, es:

```
def b(i, q):
  """
  El máximo valor total que puede obtener la biblioteca cuando
  su presupuesto es q, si compra únicamente artículos menores
  o iguales a i, y volúmenes que no tienen artículos mayores a i.
  """
  if q < 0: return -infty
  if i < 0: return 0
  if i == n - 1 or e[i] != e[i + 1]:
    j = e[i]
    return max(b(i - 1, q),
               b(i - 1, q - c[i]) + v[i],
               b(i - l[j], q - p[j]) + w[j])
  return max(b(i - 1, q), b(i - 1, q - c[i]) + v[i])
```

La respuesta al enunciado es, entonces, $b(n-1, q)$.

```
from itertools import groupby
n = 5 # Número de artículos.
m = 3 # Número de volúmenes.
q = 10 # Presupuesto.
v = [4, 5, 3, 2, 4] # Valores de cada artículo.
c = v # Costo de cada artículo.
e = [0, 0, 1, 2, 2] # Volumen al que pertenece cada artículo.
p = [7, 4, 4] # Costo de cada volumen.

# Número de artículos de cada volumen.
l = list(map(lambda x: len(list(x[1])), groupby(e)))
# Valor de cada volumen, donde el valor de un volumen es la suma
# de los valores de todos sus artículos.
```

```
    w = list(map(lambda x: sum(y[0] for y in x[1]), groupby(zip(v, e),
    lambda x: x[1])))

    print(b(n - 1, q))
    >>> 12
```

Análisis asintótico

Para analizar el número de llamadas en el peor caso, nos preguntamos cuál es el peor caso. Queremos un árbol de subproblemas que tenga muchos vértices. Para esto vamos a definir una noción de tamaño de un subproblema. La noción de tamaño acá es $i * \max(q, 0)$. En nuestro análisis vamos a usar sólo casos donde $q \geq 0$, entonces el tamaño del problema (i, q) va a ser simplemente iq.
Cada vez que vamos de un subproblema a un sub-subproblema, el tamaño del sub-subproblema es menor que el del subproblema. Cuando llegamos a un subproblema de tamaño cero, no hay más recursión. Luego, **queremos que el tamaño de nuestros subproblemas decrezca lo más lentamente posible**. Esto resultará en un árbol de subproblemas lo más lleno de vértices posible. Esto es precísamente el concepto de "peor caso", para número de llamadas: tener el máximo número de llamadas posibles, para un (n, q) dado.

En la tercer rama de nuestra función recursiva hacemos 3 llamados, mientras que en la cuarta hacemos 2. Si queremos maximizar el número de subproblemas computados, queremos entrar siempre en la tercer rama. Luego, hacemos que $e_i \neq e_{i+1}$ para todo i. Es decir, cada artículo está en un volumen distinto. Luego, como l_j es el número de artículos en el j-ésimo volumen, tenemos que $l_j = 1$ para todo j. Asimismo, como de un problema de tamaño iq vamos a un problema de tamaño $(i - l_j)(q - p_j) = (i - 1)(q - p_j)$, queremos hacer que $p_j = 0$ para todo j, decreciendo lo mínimo posible. En la rama donde devolvemos `b(i - 1, q - c[i])`, el tamaño del subproblema decrece de iq a $(i - 1)(q - c_i)$. Luego, para minimizar el decrecimiento, consideremos casos donde $c_i = 0$ para todo i.

En esta familia de casos, entonces, al tener $c_i = 0 \; \forall \; i$, $p_j = 0 \; \forall \; j$, y $l_j = 1 \; \forall \; j$, los tres llamados que vamos a hacer desde el problema (i, q) son $(i-1, q)$, $(i-1, q)$, y $(i-1, q)$. Luego, estos casos no cambian q nunca (lo cual es obvio, todos los artículos y todos los volúmenes cuestan cero, luego nuestro presupuesto nunca baja), y expande tres veces cada vez. Vemos que el árbol de recursión tiene exactamente $\frac{3^{n+1}-1}{2}$ vértices para esta familia de casos, puesto que el número de vértices en el i-ésimo nivel es 3^i, y luego la suma de todos los vértices en todos los niveles es la conocida serie geométrica $\sum_{i=0}^{n} b^i = \frac{b^{n+1}-1}{b-1}$, con $b = 3$.

Concluímos entonces que en el peor caso, nuestro algoritmo resuelve $\Omega(3^n)$ subproblemas, gastando $\Omega(1)$ operaciones en cada uno, y luego usa $\Omega(3^n)$ operaciones.

Mientras tanto, vemos que hay sólo nq posibles subproblemas **distintos**. Tenemos, entonces, muchos subproblemas compartidos, puesto que $\frac{nq}{3^n} \to 0$ cuando $n \to \infty$. Esto nos sugiere usar programación dinámica.

Programación dinámica

Podemos simplemente usar un cache, para convertir esto en programación dinámica top-down. Recordemos que hacer programación dinámica top-down, dada una función recursiva, es un proceso absolutamente mecánico, y no hay nada que pensar. Si la llamada que nos piden está en el cache, devolvemos eso. Si no, justo antes de devolver un valor, lo guardamos en el cache. Para los casos bordes (`q < 0` y `i < 0`) (las primeras dos líneas de la función) da lo mismo guardarlo en el cache que no, y en general mejor no guardarlos porque no

```
def dp_td_b(i, q, cache = {}):
  if q < 0: return -infty
  if i < 0: return 0
  if (i, q) not in cache:
    if i == n - 1 or e[i] != e[i + 1]:
      j = e[i]
      cache[(i, q)] = max(dp_td_b(i - 1, q),
                          dp_td_b(i - 1, q - c[i]) + v[i],
                          dp_td_b(i - l[j], q - p[j]) + w[j])
    else:
      cache[(i, q)] = max(dp_td_b(i - 1, q),
                          dp_td_b(i - 1, q - c[i]) + v[i])
  return cache[(i, q)]
```

Donde llamamos a `dp_td_b(n - 1, q)` para resolver el problema.

Complejidad top-down

Analizar la complejidad de un algoritmo que usa programación dinámica top-down no es totalmente trivial. Al llamar a `dp_td_b(n - 1, q)`, va a haber un árbol de recursión. Los vértices de ese árbol son o hojas, o tienen hijos (es decir, hacen llamadas recursivas). Sólo hay llamadas recursivas cuando `(i, q) not in cache`, entonces va a haber a lo sumo $n(q+1)$ de estos ($q+1$ porque podemos tener presupuestos desde 0 hasta q, inclusive, en estas llamadas). En esas llamadas, hacemos un número constante de operaciones, más el costo de buscar y escribir en el cache, al que vamos a llamar h. Luego el costo de las llamadas recursivas es $O(1) + h = O(h)$. El costo de las hojas es el costo de buscar en el cache, luego es h.
Cada llamada recursiva tiene a lo sumo 3 hijos. Como hay a lo sumo hay nq llamadas recursivas distintas, y nunca llamamos a la misma llamada recursiva dos veces porque tenemos un cache, va a haber a lo sumo $3nq$ hojas.
Luego, el costo temporal del algoritmo es $3nqh + O(h)nq = O(nqh)$.
En general, h no va a ser constante, a menos que tengamos una estructura particular donde guardar y leer estos resultados. Decir "bueno tiro todo en un hash table" no funciona, a menos

que digamos explícitamente cuál es nuestra función de hash, el tamaño de nuestra tabla, y el costo de construirla. En este caso, podemos guardar los resultados en una matriz de n filas y q columnas, que nos cuesta $O(nq)$ tiempo y espacio, y nos da lectura y escritura en tiempo $h = O(1)$.

Luego, la complejidad temporal y espacial de la solución con programación dinámica top-down es $O(nq)$.

Bottom-up

En vez de hacer llamadas recursivas, podemos llenar el cache explícitamente. Esto se llama programación dinámica bottom-up. Vemos que en cada llamada en nuestra función recursiva, para resolver el problema (i, j), usamos la solución a algún subproblema (k, r) con $k < i$, y $r \leq j$. Luego, podemos computar las entradas de este cache con dos ciclos. El ciclo exterior itera todos los i, crecientes. Dentro de cada i, iteramos todos los q, crecientes. Esto nos garantiza que para cuando tenemos que llenar la celda (i, j), ya tenemos llenadas las celdas (k, r) para todo $k < i, r < j$.

```
def dp_bu_b(n, q):
  dp = [[0 for _ in range(q + 1)] for i in range(n)]
  for i in range(1, n):
    for qq in range(q + 1):
      dp[i][qq] = dp[i - 1][qq]
      if qq >= c[i]: dp[i][qq] = max(dp[i][qq],
                                     dp[i - 1][qq - c[i]] + v[i])
      if i == n - 1 or e[i] != e[i + 1]:
        j = e[i]
        if qq >= p[j]: dp[i][qq] = max(dp[i][qq],
                                       dp[i - l[j]][qq - p[j]] + w[j])
  return dp[n - 1][q]
```

La respuesta al ejercicio se obtiene llamando a `dp_bu_b(n, q)`. Notar como combinamos la cuarta rama con la tercer rama, puesto que son muy similares, y en esta versión del código es simple hacerlo.

Esta forma de programación dinámica nos deja muy fácilmente analizar el comportamiento asintótico de nuestro programa. Usamos siempre $\Theta(nq)$ tiempo, y $\Theta(nq)$ espacio.