UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Algorítmica (11593)
Primer Parcial
8 de noviembre de 2016

etsinf
Escuela Técnica Superior de Ingeniería Informática

NOMBRE:

1 — 2.5 puntos

Estamos organizando una excursión en canoa. Debemos embarcar un total de N pasajeros y queremos utilizar el menor número de canoas. Todas las canoas son iguales y admiten una o dos plazas, pero el peso máximo que soportan es de K kilos. Para facilitarnos la tarea, tenemos a los pasajeros ordenados por su peso $w_1 \leq w_2 \leq \cdots \leq w_N \leq K$. Por ejemplo, si tenemos canoas que soportan hasta 150Kg y 4 pasajeros de pesos 52Kg, 74Kg, 91Kg y 120Kg, podemos utilizar 3 canoas.

Nos piden abordar este problema con una estrategia voraz. Para ello nos piden definir la función *canoas* que podría utilizarse como en esta llamada: `canoas([52,74,91,120],150)` que devolvería `3`.

¿Cuál es el coste temporal del algoritmo que has desarrollado?

Solución:

Nota: vamos a razonar que la estrategia propuesta es óptima aunque esto no se exigía en el enunciado.

Si nos fijamos en la persona más pesada vemos que hay 2 posibilidades: Tiene que ir solo en una canoa o puede ir acompañado. Si acompañado de la persona más ligera ya no cabe, no cabrá con ninguna otra. Por otra parte, si cabe con la persona más ligera, quizás se podría sentar con otra persona algo más pesada. ¿Es óptimo sentar entonces al más pesado con la persona más ligera?

Supongamos que en una solución óptima el más pesado se sienta con otro que no es el más ligero. Podemos intercambiar a ese por el más ligero y seguiría siendo una solución óptima. Para ver que este intercambio es posible supongamos que w_N se sentaba con w_x y que w_1 se sentaba con w_y. Tras intercambiar w_x con w_1 es obvio que $w1 + w_N \leq K$ ya que al lado de w_N hemos puesto a alguien que pesa menos. Por otra parte, w_x cabía en la canoa junto a w_N y ahora se sienta con w_y que pesa menos que w_N, con lo que también cabrán en una canoa.

Una posible solución en Python 3 de esta estrategia voraz sería como sigue:

```
def canoas(pesos, K):
  i,j,cont = 0,len(pesos)-1,0
  while i<j:
    if pesos[i]+pesos[j]<=K:
      i += 1 # metemos al flaco tb
    j -= 1
    cont+=1
  if i==j: # sobra uno al final
    cont+=1
  return cont
```

El coste temporal de este algoritmo es $O(N)$ porque el coste de cada iteración es constante y en cada una se reduce en 1 o en 2 el número de pasajeros a considerar.

Nota: Existen otras soluciones voraces óptimas. Por ejemplo, si sentamos al más pesado con el más pesado de los que cabrían con él (sería el $w_x = max\{w_i | w_i + w_N \leq K\}$), también encontraremos una solución óptima. El razonamiento para ver que es óptima es similar: si hay una solución óptima en la que el más pesado estaba solo, sentarlo con x no empeora la solución (si x estaba solo, la mejora, en otro caso dejamos solo a su acompañante y las canoas no varían). Y si el más pesado estaba sentado con otro, éste pesará menos que x y podemos intercambiarlos (mismo razonamiento: al poner uno que cabe menos seguirán cabiendo, y el que ahora se sienta con x pesará menos que w_N y seguirán cabiendo en una canoa).

Cómo llenar una caja con exactamente W Kgs. de peso (ni más ni menos) utilizando pesas de $w_1, \ldots, w_N$ Kgs. es un problema clásico conocido como *"Suma del subconjunto"*. Este problema se puede resolver con la llamada $F(N, W)$ a la siguiente ecuación recursiva:

$$F(i,c) = \begin{cases} c = 0 & \text{si } i = 0 \\ F(i-1,c) & \text{si } i > 0 \text{ y } w_i > c \\ F(i-1,c) \vee F(i-1,c-w_i) & \text{si } i > 0 \text{ y } w_i \leq c \end{cases}$$

Podemos obtener el siguiente algoritmo iterativo que nos dice si es posible o no llenar la caja.

```
def subset_sum(W,w):
  N = len(w)
  F = { (0,c):(c==0) for c in range(W+1) }
  for i in range(1,N+1):
    for c in range(W+1):
      F[i,c] = F[i-1,c]
      if not F[i,c] and c>=w[i-1]:
        F[i,c] = F[i-1,c-w[i-1]]
  return F[N,W]
```

Modifica la función anterior para que, en caso de haber solución, nos diga la lista de objetos que pueden llenar completamente la caja (en otro caso devolvería el valor `None`).

Solución:

Aunque se puede recuperar la secuencia de objetos utilizando punteros hacia atrás, también se puede recuperar sin utilizarlos:

```
def subset_sum(W,w):
  N = len(w)
  F = { (0,c):(c==0) for c in range(W+1) }
  for i in range(1,N+1):
    for c in range(W+1):
      F[i,c] = F[i-1,c]
      if not F[i,c] and c>=w[i-1]:
        F[i,c] = F[i-1,c-w[i-1]]
  # hasta aqui era identico al codigo del enunciado del examen
  c,path = W,None
  if F[N,W]: # si hay una forma de llenar la caja
    path = []
    for i in range(N,0,-1): # vamos viendo los objetos desde el ultimo
      if c>=w[i-1] and F[i,c-w[i-1]]: # si cabe y metiendolo hay solucion
        path.append((i-1,w[i-1])) # lo metemos, era valido meter el indice o el peso
        c -= w[i-1]
    path.reverse() # si los queremos devolver en orden
  return path
```

3 **5.5 puntos**

Deseamos asfaltar los K kilómetros de carretera que separan dos ciudades. Una empresa constructora nos ofrece un catálogo de N tipos de tramo con los que construir la carretera: cada uno tiene una longitud de L_i kilómetros y un coste de C_i euros, para $1 \leq i \leq N$. Como los tramos deben ser utilizados enteros (no se pueden dejar a medias), nos piden conseguir comprar 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 maximizar 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 la secuencia de tramos a utilizar).
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 como secuencias de tipos de tramo:

   $$X = \left\{ (x_1, \ldots, x_k) \in \{1, \ldots, N\}^\star | \sum_{i=1}^{k} L_{x_i} = K \right\}$$

   La función objetivo es $f((x_1, \ldots, x_k)) = \sum_{i=1}^{k} C_{x_i}$, 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.

   $$F(k) = \begin{cases} 0 & \text{si } k = 0 \\ \infty & \text{si } 0 < k < \text{mín}_{\forall 1 \leq i \leq N} L_i \\ \text{mín}_{1 \leq i \leq N, L_i \leq k} F(k - L_i) + C_i & \text{en otro caso} \end{cases}$$

   La llamada inicial sería $F(K)$. Este problema es muy parecido al cambio de monedas sin limitación del número de monedas, excepto que el coste de usar cada moneda (aquí tramos) varía según su tipo. Por tanto, es claramente mejor la aproximación de ir tramo a tramo y no con un espacio de búsqueda que contabiliza el número de tramos de cada 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):
  F, inf = [0], 2**31
  for k in range(1,K+1):
    F.append(min((F[k-li]+ci for li,ci in zip(L,C) if li<=k),default=inf))
  return F[K]
```

4. El coste temporal y espacial del algoritmo iterativo.

   El coste espacial de este algoritmo es $O(K)$, mientras que el coste temporal es $O(NK)$.

   El coste espacial es claramente el tamaño del vector `F` que es de longitud `K+1`, mientras que el coste temporal viene dado el bucle `for k···` y por el bucle `for li,ci in zip(L,C)` que itera en los vectores `L` y `C` de talla N.