UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Algorítmica (11593)

Primer Parcial
7 de noviembre de 2017

etsinf
Escuela Técnica Superior de Ingeniería Informática

NOMBRE:

1 — 4.5 puntos

Un ladrón planea robar en las casas de una misma calle. Para no levantar sospechas decide no robar nunca en 2 casas consecutivas.

Sabemos que hay N casas, numeradas de 1 a N, y que lo que puede robar en cada casa i tiene un beneficio de $B(i)$ euros, que siempre será un valor no negativo. Se desea maximizar la ganancia del ladrón. Para ello **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. Como ayuda/sugerencia te proponemos *resolver un problema auxiliar*:

   > $R(n)$ representa el máximo beneficio que conseguiría el ladrón robando las casas entre 1 y n condicionado a que entra a robar en la número n.

   Se pide una ecuación recursiva que resuelva este problema. **También** se pide que indiques cómo resolver el problema original haciendo uso de $R(n)$.
3. El algoritmo iterativo (preferiblemente en Python3) asociado a la ecuación recursiva anterior para calcular *el mayor beneficio* (no hace falta determinar la elección concreta de casas a robar).
4. El coste temporal y espacial del algoritmo iterativo.

Solución:

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

   Podemos representar las soluciones de muchas formas. Una forma podría ser un vector de talla N con 0s y 1s indicando si el ladrón roba o no en cada casa. También se puede representar mediante una lista de los números de las casas en los que el ladrón va a robar:

   $$X = \{(x_1, \ldots, x_k) \in \{1, \ldots, N\}^\star \mid x_i + 1 < x_{i+1} \forall 1 \leq i < k\}$$

   La función objetivo es $f((x_1, \ldots, x_k)) = \sum_{i=1}^{k} B(x_i)$, mientras que la solución óptima buscada es $\hat{x} = \text{argmax}_{x \in X} f(x)$.
2. Como ayuda/sugerencia te proponemos *resolver un problema auxiliar*:

   > $R(n)$ representa el máximo beneficio que conseguiría el ladrón robando las casas entre 1 y n condicionado a que entra a robar en la número n.

   Se pide una ecuación recursiva que resuelva este problema. **También** se pide que indiques cómo resolver el problema original haciendo uso de $R(n)$.

   Si el ladrón termina robando la casa n-ésima se llevará seguro el beneficio $B(n)$. Adicionalmente puede robar antes casas terminado como mucho en la casa $n-2$ siempre que éstas existan, así que una primera solución naíf podría ser:

   $$R(n) = \begin{cases} B(n) & \text{si } n \leq 2 \\ B(n) + \text{máx}_{1 \leq m \leq n-2} R(m) & \text{en otro caso} \end{cases}$$

Obviamente, la solución al problema se consigue suponiendo que el ladrón puede terminar de robar en cualquiera de las casas, de nuevo una solución algo naíf:

$$\max_{1\leq n\leq N} R(n)$$

Existe una solución más eficiente si nos damos cuenta de que no hace falta ver la ganancia de todas las casas menor o igual que $n-2$. En concreto, no hace falta mirar ni $n-4$ ni las anteriores ya que las podemos elegir y luego añadir otra en medio que será $n-2$ o $n-3$ que, al ser $B(i)\geq 0$, no hay motivo para no incluirlas, con lo que tenemos la siguiente ecuación recursiva:

$$R(n)=\begin{cases} B(n) & \text{si } n\leq 2 \\ B(n)+\max_{max(1,n-3)\leq m\leq n-2} R(m) & \text{en otro caso}\end{cases}$$

Respecto a la solución al problema, no vale la pena mirar en $N-2$ o anteriores. En particular, si el máximo estuviera en $N-2$ se podría robar también en N aumentando así su beneficio:

$$\max\{R(N-1),R(N)\}$$

(para $N\geq 2$, ya que para $N=1$ es trivialmente $R(1)=B(1)$)

3. El algoritmo iterativo (preferiblemente en Python3) asociado a la ecuación recursiva anterior para calcular *el mayor beneficio* (no hace falta determinar la elección concreta de casas a robar).

   Suponemos que `B` es un diccionario python indexado entre `1` y `N`. El algoritmo que resolvería la primera versión de la ecuación recursiva podría ser:

```
def max_roba(B):
  N = len(B)
  R = {}
  for n in range(1,N+1):
    R[n] = B[n] + max((R[m] for m in range(1,n-1)),default=0)
  return max(R[n] for n in range(1,N+1))
```

   Mientras que el código asociado a la segunda versión sería:

```
def max_roba(B):
  N = len(B)
  if N==1:
    return B[1]
  R = {}
  for n in range(1,N+1):
    R[n] = B[n] + max((R[m] for m in range(max(1,n-3),n-1)),default=0)
  return max([R[N-1],R[N]])
```

4. El coste temporal y espacial del algoritmo iterativo.

   El coste espacial de este algoritmo es $O(N)$ en los dos casos, mientras que el coste temporal es $O(N^2)$ en el primer caso y $O(N)$ en el segundo.P(n-1)

Nota: Aunque no se pedía resolverlo así, también es posible resolver este problema, *de nuevo en el caso en que suponemos* $B(n)>0\ \forall n$, resolviendo directamente el problema original $P(n)$ (y no el subproblema $R(n)$) con la siguiente ecuación recursiva:

$$P(n)=\begin{cases} B(n) & \text{si } 1\leq n\leq 2 \\ \max\{P(n-1),P(n-2)+B(n)\} & \text{si } n>2\end{cases}$$

Que corresponde a la siguiente versión iterativa en la que, si no nos piden la secuencia de casas que hay que robar, se puede resolver con reducción del coste espacial (coste espacial constante, el coste temporal sigue siendo lineal):

```
def maxroba4(B):
  N = len(B)
  P = [B[1],B[2]]
  for n in range(3,N+1):
    P[n % 2] = max(P[(n-1) % 2], P[(n-2) % 2] + B[n])
  return P[N % 2]
```

2 **2.5 puntos**

El siguiente código calcula el mínimo número de monedas necesario para devolver la cantidad `Q` usando `N` tipos de moneda siendo `v` y `m` vectores de longitud `N` que contienen, respectivamente, el valor y la cantidad de monedas de cada tipo:

```
def number_of_coins(Q, v, m, infinity=2**31):
  N = len(v) # numero de tipos de moneda, N==len(m) también
  M = {} # M[i,q] es el numero de monedas para representar cantidad q
         # usando los i primeros tipos de moneda
         # con 0 tipos de moneda solamente podemos representar la cantidad 0
  M[0,0] = 0
  for q in range(1, Q+1):
    M[0,q] = infinity
  # para más de 0 tipos de moneda, desde 0 hasta N inclusive:
  for i in range(1,N+1):
    for q in range(Q+1): # probar cualquier cantidad entre 0 y Q inclusive
      M[i,q] = min(M[i-1,q-x*v[i-1]]+x for x in range(min(q//v[i-1], m[i-1])+1))
  return M[N,Q]

Q = 24
v = [1, 2, 5, 10, 20, 50]
m = [3, 1, 4,  1,  2,  1]
print(number_of_coins(Q, v, m))
```

Se pide: que realices los cambios necesarios que consideres para que la función devuelva, en lugar del mínimo número de monedas utilizada, las monedas que consiguen ese mínimo en forma de un vector de talla `N` donde en la posición i-ésima esté el número de monedas a utilizar de valor `v[i]`.

Solución:
Una posible solución utilizando *backpointers*:

```
def number_of_coins_path(Q, v, m, infinity=2**31):
  N = len(v) # numero de tipos de moneda, N==len(m) también
  M,B = {},{} # M[i,q] es el numero de monedas para representar
              # cantidad q usando los i primeros tipos de moneda con 0
              # tipos de moneda solamente podemos representar la
              # cantidad 0
  M[0,0] = 0
  for q in range(1, Q+1):
    M[0,q] = infinity
  # para más de 0 tipos de moneda, desde 0 hasta N inclusive:
  for n in range(1,N+1):
    for q in range(Q+1): # probar cualquier cantidad entre 0 y Q inclusive
      M[n,q],B[n,q] = min((M[n-1,q-x*v[n-1]]+x,x) for x in range(min(q//v[n-1], m[n-1])+1))
  cambio = []
  n,q = N,Q
  while (n>0):
    cambio.append(B[n,q])
    q -= v[n-1]*B[n,q]
    n -= 1
  cambio.reverse()
  return M[N,Q],cambio
```

Aunque también es posible recuperar el camino sin hacer uso de los *backpointers*:

```
def number_of_coins_path2(Q, v, m, infinity=2**31):
  N = len(v) # numero de tipos de moneda, N==len(m) también
  M = {} # M[i,q] es el numero de monedas para representar cantidad q
         # usando los i primeros tipos de moneda con 0 tipos de moneda
         # solamente podemos representar la cantidad 0
  M[0,0] = 0
  for q in range(1, Q+1):
    M[0,q] = infinity
  # para más de 0 tipos de moneda, desde 0 hasta N inclusive:
  for n in range(1,N+1):
    for q in range(Q+1): # probar cualquier cantidad entre 0 y Q inclusive
      M[n,q] = min(M[n-1,q-x*v[n-1]]+x for x in range(min(q//v[n-1], m[n-1])+1))
  cambio = []
  n,q = N,Q
  while (n>0):
    aux,usadas = min((M[n-1,q-x*v[n-1]]+x,x) for x in range(min(q//v[n-1], m[n-1])+1))
    cambio.append(usadas)
    q -= v[n-1]*usadas
    n -= 1
  cambio.reverse()
  return M[N,Q],cambio
```

3 **3 puntos**

Una pizzería tiene una lista de ingredientes y desea enumerar todas las posibles formas de combinar 4 de esos ingredientes de manera que no aparezcan simultáneamente dos ingredientes de una lista de ingredientes que no combinan:

```
ingredientes = ['mozzarela','anchoa','alcaparra','jamon','atun',
                'champinyon','aceituna','chorizo']
no_combinan = [('mozzarela','anchoa'),('atun','chorizo'),('jamon','atun'),
               ('anchoa','chorizo'),('anchoa','champinyon')]
```

La lista de ingredientes es una lista Python de cadenas, mientras que la lista `no_combinan` está representada por una lista de tuplas donde cada tupla tiene 2 ingredientes en el mismo orden relativo que aparecen en la lista `ingredientes` (por ejemplo, aparece la tupla `('atun','chorizo')` donde `'atun'` aparece antes en `ingredientes` que `'chorizo'`).

Se pide: utilizar la estrategia de búsqueda con retroceso o *backtracking* para mostrar por salida estándar todas las combinaciones de 4 ingredientes que combinen. Por ejemplo, la llamada:

```
listar(ingredientes,no_combinan)
```

produciría el siguiente resultado por pantalla:

```
mozzarela, alcaparra, jamon, champinyon
mozzarela, alcaparra, jamon, aceituna
mozzarela, alcaparra, atun, champinyon
mozzarela, alcaparra, atun, aceituna
mozzarela, alcaparra, champinyon, aceituna
mozzarela, jamon, champinyon, aceituna
mozzarela, atun, champinyon, aceituna
anchoa, alcaparra, jamon, aceituna
anchoa, alcaparra, atun, aceituna
alcaparra, jamon, champinyon, aceituna
alcaparra, atun, champinyon, aceituna
```

Como pista comentar que para mostrar `['a','b','c']` como arriba se puede utilizar:

```
>>> print(", ".join(['a','b','c']))
a, b, c
```

y para sacar la posición de un ingrediente en la lista se puede utilizar el método `index`:

```
>>> ingredientes.index('atun')
4
```

Es **importante** no sacar varias veces por pantalla una misma combinación de ingredientes (incluso si aparecen en un orden diferente).

Nota: Aunque no es la solución más deseable, si te resulta más fácil puedes suponer que en lugar de una lista de ingredientes se proporciona el número de ingredientes N y que la lista de ingredientes que no combinan hace referencia a ellos usando enteros entre 0 y $N-1$. De manera que

```
listarN(7,[(0,1),(4,7),(3,4),(1,7),(1,5)])
```

produciría la siguiente salida:

```
[0, 2, 3, 5]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[0, 2, 5, 6]
[0, 3, 5, 6]
[0, 4, 5, 6]
[1, 2, 3, 6]
[1, 2, 4, 6]
[2, 3, 5, 6]
[2, 4, 5, 6]
```

Solución:

```
def listar(ingredientes,no_combinan):
  def prometedor(node,nuevo):
    for a,b in no_combinan:
      if b==nuevo and a in node:
        return False
    return True
  def backtracking(node):
    if len(node)==4:
      print(", ".join(node))
    else:
      primero = 0 if len(node)==0 else ingredientes.index(node[-1])+1
      for nuevo in ingredientes[primero:-1]:
        if prometedor(node,nuevo):
          backtracking(node+[nuevo])
  backtracking([])
```

La solución del problema simplificado es muy parecida:

```
def listarN(N,no_combinan):
  def prometedor(node,nuevo):
    for a,b in no_combinan:
      if b==nuevo and a in node:
        return False
    return True
  def backtracking(node):
    if len(node)==4:
      print(node)
    else:
      primero = 0 if len(node)==0 else node[-1]+1
      for nuevo in range(primero,N):
        if prometedor(node,nuevo):
          backtracking(node+[nuevo])
  backtracking([])
```