UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escuela Técnica Superior de Ingeniería Informática

Algorítmica (11593)-P1

03 de noviembre de 2014

1 — 2 puntos

Una *fracción egipcia* es una forma de representar un número racional como una suma de fracciones unitarias (aquellas donde el numerador es igual a 1). Por ejemplo:

$$\frac{5}{6} = \frac{1}{2} + \frac{1}{3}$$

El matemático Fibonacci propuso un algoritmo voraz para obtener esta expresión para cualquier número racional extrayendo la fracción unitaria de mayor valor y aplicando la misma idea al resto:

$$\frac{x}{y} = \frac{1}{\lceil y/x \rceil} + \frac{(-y) \bmod x}{y \lceil y/x \rceil}$$

Por ejemplo, aplicado a 7/15 nos daría:

$$\frac{7}{15} = \frac{1}{\lceil 15/7 \rceil} + \frac{(-15) \bmod 7}{15 \lceil 15/7 \rceil} = \frac{1}{3} + \frac{6}{45} = \cdots = \frac{1}{3} + \frac{1}{8} + \frac{1}{120}$$

Se pide implementar la siguiente función python que recibe como argumentos el numerador y denominador de un número racional y devuelve una lista de los denominadores de las fracciones unitarias:

```
>>> egyptian(7,15)
[3, 8, 120]
```

Nota: el redondeo al alza de un número con decimales en python se puede conseguir con la función `ceil` de la biblioteca `math`.

Solución:

```
import math
def egyptian(x,y):
  resul = []
  while x != 0:
    aux = math.ceil(y/float(x))
    resul.append(aux)
    x,y = -y % x, y*aux
  return resul
```

2 — 2 puntos

María quiere matricularse a una serie de cursos entre unos días determinados y obtener el máximo número de créditos. Cada curso tiene una fecha de inicio y otra de finalización. Como todos los cursos dan el mismo número de créditos y María ha cursado algorítmica, tiene claro qué algoritmo aplicar para elegir los cursos. Juan también quiere matricularse, pero él no tiene restricción en los días aunque decide matricularse de los mismos cursos que María y de todos los que pueda para maximizar sus créditos. Juan no sabe los cursos a los que se matricula su amiga, pero por suerte también ha estudiado algorítmica y conoce el algoritmo que aplicará:

```
def seleccion_act(C):
  x  = set()
  t1 = min(s for (s,t) in C)
  for (s,t) in sorted(C,key=lambda (s,t):t):
    if (t1<=s):
      x.add( (s,t) )
      t1 = t
  return x
```

A partir de estos datos:

- `C` es la lista de tuplas `(inicio,fin)` de los cursos,
- `(iniMaria,finMaria)` el intervalo en que María se puede matricular

y haciendo uso de `seleccion_act`, **se pide:**

- código python que calcule la lista de cursos en que se matricula María,
- código python que calcule la lista de cursos en que se matricula Juan, explicando brevemente la estrategia utilizada por Juan para maximizar sus propios créditos sujeto a asistir a los mismos cursos que su amiga. Puedes asumir que existe una función llamada `solapan(x,y)` que recibe dos intervalos `x` e `y` (son tuplas `(inicio,fin)`) y que devuelve un booleano indicando si ambos intervalos solapan o no.

Solución:

```
def solapa((s1,e1),(s2,e2)):
  return s1<e2 and s2<e1

def selMaria(C,(iniMaria,finMaria)):
  CMaria = [(s,t) for (s,t) in C if iniMaria<=s and t<=finMaria]
  print "CMaria",CMaria
  return seleccion_act(CMaria)

# una manera facil de entender
def selJuan(C,(iniMaria,finMaria)):
  cursosMaria = selMaria(C,(iniMaria,finMaria))
  Caelegir = [c for c in C if not any(solapa(c,x) for x in cursosMaria)]
  adicionales = seleccion_act(Caelegir) if len(Caelegir)>0 else []
  return cursosMaria.union(adicionales)

# una manera mas eficiente si tenemos en cuenta que en el cjt de
# actividades de Maria no hay solapes ni caben otras actividades:
def selJuan(C,(iniMaria,finMaria)):
  cursosMaria = selMaria(C,(iniMaria,finMaria))
  if len(cursosMaria>0):
    minMaria = min(s for (s,t) in cursosMaria)
    maxMaria = max(t for (s,t) in cursosMaria)
    Caelegir = [(s,t) for (s,t) in C if t<=minMaria or s>=maxMaria]
  else:
    Caelegir = C
  adicionales = seleccion_act(Caelegir) if len(Caelegir)>0 else []
  return cursosMaria.union(adicionales)
```

3 — 2 puntos

El siguiente algoritmo iterativo de Programación Dinámica resuelve el problema del cálculo de la probabilidad del trayecto más probable en el río Congo:

```
def iterative_maximum_probability(E, p):
   P = [None]*E
   P[0] = 1
   P[1] = p(1,0)
   for i in xrange(2, E):
      P[i] = max( P[i-1] * p(i,i-1), P[i-2] * p(i,i-2) )
   return P[E-1]
```

siendo `E` el número de embarcaderos y `p(i,j)` una función que devuelve la probabilidad de llegar al embarcadero `i` desde `j` (ambos numerados de `0` a `E-1`). Responde a las siguientes cuestiones:

a) Analiza el coste temporal y espacial del algoritmo.
b) Modifica el código para conocer también el camino de máxima probabilidad.

Solución:

```
def iterative_maximum_probability(E, p):
   P,B = [None]*E,[None]*E
   P[0] = 1
   P[1] = p(1,0)
   for i in xrange(2, E):
      P[i],B[i] = max( (P[i-1] * p(i,i-1),i-1), (P[i-2] * p(i,i-2),i-2) )
   path = [E-1]
   while B[path[-1]] != None:
     path.append(B[path[-1]])
   path.reverse()
   return P[E-1],path
```

4 — 4 puntos

Tras una catástrofe natural, un continente necesita ayuda humanitaria (víveres, material médico). Se han preparado paquetes *idénticos* con este tipo de ayuda y se deben distribuir a las distintas zonas. Se tienen P paquetes y se plantea cómo distribuirlos entre las Z zonas afectadas. Se ha estimado el beneficio a conseguir en cada zona z, para z entre 1 y Z, dependiendo del número de paquetes p que se envien. Esta estimación se obtiene con la función $b(z, p)$. Los expertos también han decidido que, como mínimo, a cada zona se le deben enviar n paquetes.

En esta situación, deseamos calcular la distribución de los paquetes entre las zonas que maximice el beneficio total. Para ello nos piden:

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 de Programación Dinámica que resuelva el problema de encontrar el máximo beneficio que se puede alcanzar. Indica cuál sería la llamada inicial para resolver el problema.

3. El algoritmo iterativo asociado a la ecuación recursiva anterior para calcular *el beneficio*.

4. Analiza el coste temporal y espacial del algoritmo iterativo.

5. ¿Se puede reducir el coste espacial y/o temporal del algoritmo anterior? ¿En qué condiciones?

Solución:

Este ejercicio se puede resolver, al menos de dos maneras diferentes:

- Modificando el problema de asignación de recursos tal cual se ha visto en teoría para añadir un número mínimo de unidades a cada zona.
- Repartiendo *a priori* los paquetes mínimos entre las zonas y usando el problema estándar de asignación de recursos para repartir el resto.