Laboratorio 6 - Algoritmia y Complejidad

Alejandro Madrazo

20 de Septiembre de 2018

1 Problema

Algoritmos Codiciosos
Fracciones Egipcias:

Cada fraccion positiva se puede representar como una suma de fracciones unitarias. Fracciones unitarias son aquellas donde el numerador es 1 y el denominador es un entero positivo.

Para una fraccion de forma nume/deno donde deno > nume obtengan la mayor fraccion unitaria posible, y recurran hasta encontrarla.
Desarrollen un algoritmo codicioso que resuelva las fracciones egipcias y describan por que es codicioso.

Respuesta:

Algorithm 1 Fracciones Egipcias

```
1:  procedure EGYPT(a, d)                    ▷ a=numerador; d=denominador
2:      if (a > d) then
3:          return "Fraccion invalida"
4:      else:
5:          c = cieling(d/a)
6:          print("1/" + C)
7:          a = a * c − d
8:          d = d * c
9:          if (n == 0) then
10:             return "Fin..."
11:         else:
12:             Egypt(a, d)
13:
```

Este es un algoritmo codicioso, ya que utiliza el resultado más optimo en cada paso. Esto lo podemos nodar en la linea número 5, en donde se elige la fracción más grande posible.

2 Problema

Algoritmos Codiciosos + Dinamicos

Desarrollen un programa dinamico y otro codicioso para el problema de knapsack fraccionario. Encontrando el valor maximo.

Algoritmo Dinámico y a la vez codicioso

En este algoritmo asumimos que los items a robar, están ordenados de mandera descendiente y que se peuden dividir en unidades.
Entonces: el arreglo Items[ ] representa el peso en la posición de la unidad. Ejemplo: Si tenemos 5 oros con valor 0.25 y 3 platas con valor 0.20 cada uno
Items = [0.25/0.25/0.25/0.25/0.25/0.20/0.20/0.20]

Algorithm 2 KnapSack

```
1:  procedure KnapSack(Items[], maxW)                          ▷
2:      knapSack[ ]
3:      w = 0                                  ▷ El peso inicial robado comienza en 0
4:      knapSack[ ].len = maxW    ▷ Nuestra 'mochila' tiene le largo de maxW
5:      for (i in items.len) do
6:          if ((w + items[i]) <= MaxW) then
7:              KnapSack[i] = items[i]
8:              w = w + items[i]
9:          else:
10:             jump
11:
```

El algoritmo anterior resulta ser dinámico al usar un arreglo para guardar lo que llevamos "robado" y otro para analizar las unidades disponibles. A la véz en cada iteración se elige primero los metales mas valiosos en relación a su peso y de último se llenan los menos valiosos. Lo cual lo hace también codicioso.

Algoritmo Codicioso

En el siguiente algoritmo asumimos que existe un arreglo llamado Items[] el cual contiene la cantidad de metal disponible en cada posición ordenados de mas valioso a menos valioso.
Ejemplo: Items [30/20/10]
Siendo Items[1] = Oro, Items[2] = Plata y Items[3] = Cobre

Algorithm 3 KnapSack

```
1: procedure KnapSack(Items[], maxW)                                    ▷
2:     knapSack[ ] = Items.len          ▷ Nuestra mochila tiene el tamaño de la
   cantidad de categorias.
3:     for (i in items.len) do
4:         if (maxW >= items[i]) then
5:             KnapSack[i] = items[i]
6:             maxW = maxW - items[i]
7:         else:
8:             Knapsack[i] = maxW
9:             maxW = 0
10:
```