Laboratorio 6

José Alejandro Guzmán Zamora

20 de septiembre de 2018

- Algoritmos Codiciosos
  Para una fracción de forma *nume/deno* en la que *deno* > *nume* obtenga la mayor fracción unitaria posible, y recurra hasta encontrarla. Desarrolle un algoritmo codicioso que resuelva las fracciones egipcias y describa por qué es codicioso.

  El problema de las fracciones egipcias se puede resolver de una manera exageradamente complicada o, por el contrario, aplicando programación codiciosa. En el caso que se desee resolver el problema de la manera más extensa, se puede considerar todas las combinaciones de sumas de fracciones unitarias hasta llegar al resultado deseado. No obstante, al visualizar algunos ejemplos se puede llegar a una solución más eficiente. Si se empieza con la fracción original, se pueden hacer restas para verificar que fracciones unitarias suman la misma. En este caso la subestructura óptima se encuentra al restar, si la resta entre la fracción original y la fracción unitaria temporal no es negativa, el resultado obligatoriamente puede seguir restando fracciones unitarias, a menos a que sea el mismo una fracción unitaria. En este caso, la decisión codiciosa que se hace es restarle al valor original la fracción unitaria con el siguiente denominador $(1/2, 1/3, 1/4, 1/5...)$ siempre y cuando el resultado no sea negativo porque en ese caso la fracción unitaria es mayor a la original.

ultima_unitaria_recursivo(number, last)

```
1       if number.numerator == 1:
2           return number
3       else:
4           temporal = number
5           number = number - 1/last
6           if number > 0:
7               ultima_unitaria_recursivo(number, last + 1)
8           else:
9               ultima_unitaria_recursivo(temp, last + 1)
```

El algoritmo mostrado tiene 'tail recursion', por lo tanto, se puede expresar

iterativamente de una manera sencilla:

---

ultima_unitaria_iterativo(number)

```
1       contador = 2
2       while number.numerator != 1:
3           temporal = number
4           number = number - 1/contador
5           contador += 1
6           if number < 0:
7                   number = temporal
8       return number
```

En pocas palabras, el algoritmo es codicioso porque en cualquier iteración la resta de la "siguiente"fracción unitaria lo lleva a verificar si la misma pertenece a la representación egipcia.

- Algoritmos Codiciosos + Dinámicos
  Desarrolle un programa dinámico y otro codicioso para el problema de knapsack fraccionario. Encontrando el valor máximo.

  Para desarrollar el algoritmo de knapsack fraccionario, es útil pensar en una recursión que lo resuelva, sin ponerle atención a la eficiencia del mismo. El siguiente algoritmo devuelve el valor máximo que el criminal puede robar. Es importante mencionar que el algoritmo recibe un arreglo que contiene las fracciones individuales en órden de mayor a menor, basado en el valor por libra de cada elemento. Por ejemplo en el caso de los elementos de cobre, plata y oro expresados en la asignación, el arreglo tendría 10 libras de valor 6 para el cobre, 20 libras de valor 5 para la plata y 30 libras de valor 4 para el oro; todo de manera individual.

---

knapsack_inicial(valuaciones, comienzo, peso_max)

```
1       valor = 0
2       for i from comienzo to peso_max:
3           comienzo += 1
4           valor = max(valor, valuaciones[i] + knapsack_inicial(valuaciones, comienzo, peso_max)
5       return valor
```

A pesar de que este algoritmo es bastante ineficiente, se aprovecha del contexto; como el arreglo está ordenado, solamente hace recursiones a lo largo del mismo con la longitud arbitraria de peso máximo. Con una modificación ligera, es posible que el algoritmo recuerde las operaciones que se repiten:

```
memoized_knapsack(valuaciones)

resultados = []
for i from comienzo to peso_max:
    resultados[i] = 0
return knapsack_auxiliar(valuaciones, 0, resultados)
```

```
knapsack_auxiliar(valuaciones, comienzo, resultados)

temp = comienzo
if resultados[comienzo] != 0:
    return resultados[comienzo]
valor = 0
for i from comienzo to peso_max:
    comienzo += 1
    valor = max(valor, valuaciones[i] + knapsack_auxiliar(valuaciones, comienzo, resultados)
resultados[temp] = valor
return valor
```

En el algoritmo superior, al momento que el valor que se busque ya esté dentro del arreglo resultados", termina esa parte de la recursión. Durante el estudio del problema antes del intento por resolverlo, resultó clara la razón para utilizar programación codiciosa. El problema de knapsack fraccionario prácticamente se convierte en seleccionar fracciones de pesos con un valor que se repite en el caso de que sean el mismo elemento.

valuaciones = [6,6,6,6,6,6,6,6,6,6,5,5,5,5,5,5,5,5,
5,5,5,5,5,5,5,5,5,5,5,5,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4]

Es intuitivo que lo que se tiene que hacer es seleccionar la mayor cantidad de elementos del mayor valor, seguido del elemento con mayor valor después del mismo, hasta ya no tener espacio. Al momento de comparar los algoritmos, resulta prácticamente inutil siquiera intentar desarrollar un programa recursivo dinámico para este problema específico.

```
knapsack_final(valuaciones)

valor = 0
for i from 1 to peso_max:
    valor += valuaciones[i]
return valor
```