Problema 1

Algorithm 1 Algoritmos Codiciosos

```
1:  procedure FRACCIONES-EGIPCIAS(numerador, denominador)
2:      Print (numdr + ' / ' + denomdr + ' = ' )

3:      while Termina ≠ 'Si' do
4:          aprx ← Floor(numdr/denomdr)
5:          Print('1/' + (aprx + 1)) // Fraccion Unitaria
6:          res ← (numdr/denomdr) − (1/(aprx + 1)) // residuo

7:          if getNumerador(res) ≠ 1 then
8:              numdr ← getDenomdr(res)
9:              denomdr ← getNumerador(res)
10:         else
11:             Print 'Se encontró la fracción unitaria mas cercana'
12:             Termina ←' Si'
```

Problema 1

- Desarrollen un algoritmo codicioso que resuelva las fracciones egipcias y describan por que es codicioso:

Es codicioso porque en el loop va generando las fracciones unitarias una tras otra. Sin embargo, se podría descomponer cada fracción entre mas fracciones. Este algoritmo solo enfoca en su entorno, es decir, en el resultado al momento. No se preocupa por conocer si hay otra combinación posible de fracciones unitarias

Problema 2

Algorithm 2 Algoritmos Codiciosos + Dinamicos

```
1:  procedure KNAPSACK GREEDY(Valor, Peso, ArrDiv, PesMax)
2:      MaxPes ← PesMax
3:      while MaxPes > 0 do
4:          i ← GetIndex(Max(ArrDiv))
5:          if ArrDiv[i] < MaxPes then
6:              MaxPes −= ArrDiv[i]
7:              Val += Valor[i]
8:              Wgth += Peso[i]
9:              ArrDiv[i] ← 0
10:         else
11:             ArrDiv[i] −= MaxPes
12:             Val += (MaxPes/Peso[i]) ∗ Valor[i]
13:             Wgth += (MaxPes/Peso[i]) ∗ Peso[i]
14:             MaxPes −= MaxPes
15:     Print "Valor = " + Val
16:     Print "Peso = " + Wgth
```

Problema 2

Algorithm 3 Algoritmos Codiciosos + Dinamicos

```
1: procedure KNAPSACK DYNAMIC(Valor, Peso,ArrDiv, PesMax, n)
2:     K ← [[0forxinrange(W + 1)]forxinrange(n + 1)]
3:     for i InRange (n + 1) do
4:         for w InRange (w + 1) do
5:             if i == 0 or w == 0 then
6:                 K[i][W] ← 0
7:             else if wt[i − 1] ← w then
8:                 K[i][w] ← max(val[i − 1] + K[i − 1][w − wt[i − 1]], K[i − 1][w])
9:             else K[i][w] ← K[i − 1][w]
10:    return K[n][W]
```