Análisis de algoritmos : Taller 3

Carlos Escobar [1] Fabián Rojas[1] Julián Tarazona[1]

[1]Pontificia Universidad Javeriana
{escobartc, fabian-rojasm, jdtarazona}@javeriana.edu.co
March 24, 2022

1 Resumen

En este documento se presenta el problema de dar el cambio ante un pago y su solución a través del análisis, diseño e implementación de un algoritmo utilizando programación dinámica.

2 Análisis y diseño

2.1 Análisis

El problema a resolver se puede describir informalmente como dar el cambio ante el ingreso de una cantidad de dinero por un pago. Para este problema en concreto se dispone de una cantidad infinita de monedas de n denominaciones D = $\{ d_1 < d_2 < ... < d_n \}$. Las denominaciones de las monedas son números naturales, al igual que las vueltas a calcular (p,v $\in N$). Este ejercicio se debe resolver utilizando programación dinámica por lo que el primer paso es escribir la función del algoritmo:

$$m_{i,v} = \begin{cases} 0; v = 0 \\ v; i = 1 \\ m(i-1, v); d_i > v \\ min(m(i, v-d_i)+1, m(i-1), v) \end{cases}$$

2.2 Diseño

1. **Entradas:**
   - Un precio (p) que representa el valor a pagar y que p $\in \mathbb{N}$.
   - Un monto (m) que representa el valor pagado y que m $\in \mathbb{N}$.
   - Una secuencia C que contiene las denominaciones de las monedas donde $C_i \in \mathbb{N}$.
2. **Salidas:**
   - Las vueltas calculadas v para la diferencia entre el total y el monto, donde v $\in \mathbb{N}$.

3 Algoritmos

3.1 Inocente

Algorithm 1 CoinChange.

```
1: procedure CoinChange(prize,paid,C)
2:     v ← |prize − paid|
3:     return CoinChangeAux(C,v,|C|)
4: end procedure
```

Algorithm 2 CoinChangeAux.

```
1: procedure CoinChangeAux(C,v,i)
2:     if v = 0 then
3:         return 0
4:     else if i = 1 then
5:         return v
6:     else
7:         if C[i] > v then
8:             return CoinChangeAux(C,v,i − 1)
9:         else
10:            a ← CoinChangeAux(C, v − C[i], i) + 1
11:            b ← CoinChangeAux(C, v, i − 1)
12:            if a < b then
13:                return a
14:            else
15:                return b
16:            end if
17:        end if
18:    end if
19: end procedure
```

3.2 Memoizado

Algorithm 3 CoinChangeAuxM.

```
1: procedure CoinChangeM(C,prize,paid)
2:     v ← |prize − paid|
3:     let M [1...v, 1...|C|] be new table
4:     for i ← 1to v do
5:         for j ← 1to |C| do
6:             M[i, j] ← ∞
7:         end for
8:     end for
9:     return CoinChangeAuxM(C, v, i, M)
10: end procedure
```

Algorithm 4 CoinChangeAuxM.

```
1: procedure CoinChangeAuxM(C,v,i,M)
2:     if M[v,i] = ∞ then
3:         if v = 0 then
4:             M[v,i] ← 0
5:         else if i = 1 then
6:             M[v,i] ← v
7:         else
8:             if C[i] > v then
9:                 M[v,i] ← CoinChangeAuxM(C,v,i−1,M)
10:            else
11:                a ← CoinChangeAuxM(C,v−C[i],i,M)+1
12:                b ← CoinChangeAuxM(C,v,i−1,M)
13:                if a < b then
14:                    M[v,i] ← a
15:                else
16:                    M[v,i] ← b
17:                end if
18:            end if
19:        end if
20:    end if
21:    return M[v,i]
22: end procedure
```

3.3 Bottom-up

Algorithm 5 CoinChangeBU.

```
1:  procedure CoinChangeBU(C,prize,paid)
2:      v ← |prize − paid|
3:      let M[1...v, 1...|C|] be new table
4:      for i ← 1to v do
5:          for j ← 1 to |C| do
6:              M[i, j] ← 0
7:          end for
8:      end for
9:
10:     for j ← 1 to v do
11:         M[j, 0] ← j
12:     end for
13:
14:     for i ← 1 to |C| do
15:         for j ← 1 to v do
16:             if C[i] > j then
17:                 M[j, i] ← M[j, i − 1]
18:             else
19:                 a ← M[j − C[i], i] + 1
20:                 b ← M[j, i − 1]
21:                 if a < b then
22:                     M[j, i] ← a
23:                 else
24:                     M[j, i] ← b
25:                 end if
26:             end if
27:         end for
28:     end for
29:     return M[v, |C|]
30: end procedure
```

3.4 Bottom-up y backtracking

Para el algoritmo de dar el cambio en Bottom-up y backtracking se define la expresión $(A, B) \in B[ic, j]$ como el un conjunto par de números que pertenecen a la tabla B en la posición (ic, j). Cuando se utiliza la expresión $(A, -)$ o $(-, B)$ se refiere a que solo se esta tomando un solo valor del par de números, el valor a tomar se muestra con una letra en mayuscula.

Algorithm 6 CoinChangeBuB.

```
1: procedure CoinChangeBuB(C,prize,paid)
2:     v ← |prize − paid|
3:     let M[1...v, 1...|C|] be new table
4:     let B[1...v, 1...|C|] be new table
5:     for i ← 1to v do
6:         for j ← 1 to |C| do
7:             M[i, j] ← 0
8:         end for
9:     end for
10:
11:    for i ← 1to v do
12:        for j ← 1 to |C| do
13:            B[i, j] ← (0, 0)
14:        end for
15:    end for
16:
17:    for j ← 1 to v do
18:        M[j, 0] ← j
19:    end for
20:
21:    for i ← 1 to |C| do
22:        for j ← 1 to v do
23:            if C[i] > j then
24:                M[j, i] ← M[j, i − 1]
25:                B[j, i] ← (j, i − 1)
26:            else
27:                a ← M[j − C[i], i] + 1
28:                b ← M[j, i − 1]
29:                if a < b then
30:                    M[j, i] ← a
31:                    B[j, i] ← (j − C[i], i)
32:                else
33:                    M[j, i] ← b
34:                    B[j, i] ← (j, i − 1)
35:                end if
36:            end if
37:        end for
38:    end for
39:    j ← |C| and i ← v
40:    let cant[1, |C|]
41:    for k ← 1 to |C| do
42:        cant[k] ← 0
43:    end for
44:
45:    while i ≠ 0 do
46:        ic ← i
47:        i ← (A, −) ∈ B[ic, j]
48:        i ← (−, B) ∈ B[ic, j]
49:        if ic ≠ i then
50:            if j = 0 then
51:                cant[j] ← ic
52:            else
53:                cant[j] ← cant[j] + 1
54:            end if
55:        end if
56:    end while
57:    ImprimirResultado(C, cant)
58: end procedure
```

Algorithm 7 ImprimirResultado.

```
1: procedure ImprimirResultado(C,cant)
2:     for k ← 1 to |C| do
3:         if cant[k] = 1 then
4:             Print cant[k] moneda de C[k]
5:         else if cant[k] > 1 then
6:             Print cant[k] monedas de C[k]
7:         end if
8:     end for
9: end procedure
```

4 Implementación

Anexo a este documento se encuentra la implementación del algoritmo de dar el cambio bottom-up y backtracking en el lenguaje C++, en el algoritmo se incluye un experimento de mil pruebas para evaluar la calidad del algoritmo.

5 Complejidad

Por inspección de codigo podemos observar que el código tiene una complejidad de $O(n^2)$, pero el ciclo que realiza iteraciones desde 1 hasta el valor de las vueltas puede llegar a ser mucho mayor que el valor la cantidad de monedas, la complejidad del algoritmo seria de $O(nm)$.

6 Resultados obtenidos

Tras realizar la implementación se automatizó el algoritmo con el fin de probar miles de casos para así dar el cambio de monedas frente a diferentes totales de pago y montos generados "aleatoriamente". Una vez hecho esto se procedió a ejecutar la función del cálculo de vueltas mil veces con los diferentes valores de las variables ante mencionadas y, como resultado final, se obtuvo un tiempo total de 1.73 segundos aproximadamente junto con un tiempo promedio de 0.00173393 segundos todo esto para las mil ejecuciones de prueba y ver el comportamiento del algoritmo en diferentes casos. A partir de eso logramos determinar que el algoritmo ejecuta de forma relativamente rápida las operaciones necesarias para el cálculo de vueltas en los mil casos y siempre arroja resultados precisos. Sin embargo, el programa tiende a tener complicaciones cuando la generación de los números aleatorios es aún más grande de lo que se estipuló en la implementación, por lo que en ocaciones se nota un alto uso de CPU y esto conlleva a que el programa termine abruptamente. Las pruebas mencionadas inicialmente están codificadas en la implementación.