Juan Diego Cardona 201819447
Mateo Vallejo 201731797

Tarea 5

Parte 1. El problema de las vueltas

Se desea construir un algoritmo que dada una cantidad total de dinero P y unas denominaciones de monedas d_1, d_2, ... d_n, determine la cantidad mínima de monedas m_1, m_2, ..., m_n, tal que la suma total de dinero sea igual a P.

Para este problema, descargar el proyecto adjunto y realizar los siguientes pasos:

1. Formalizar el problema de las vueltas describiendo sus entradas, salidas, precondición y postcondición

Entrada/Salida	Nombre	Tipo
E	cantidad	nat
E	denom	array [0:N] nat
S	cant.M	array [0:N] nat

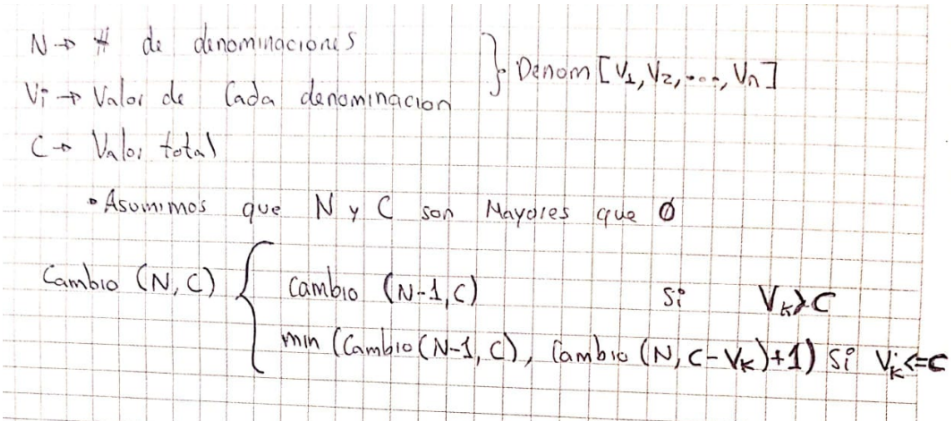
Precondición:

(N>0) ∧ (cantidad >= 0) ∧ (∀k | 0<=k<=N: denom[k]>0)

Postcondición:

(+i | i∈ denom ∧ cant.M : denom[i] •cant.M[i])=cantidad

2. Definir una función que represente el valor a minimizar y una ecuación de recurrencia que permita calcular esta función. Argumentar por qué la ecuación de recurrencia efectivamente calcula la función diseñada.



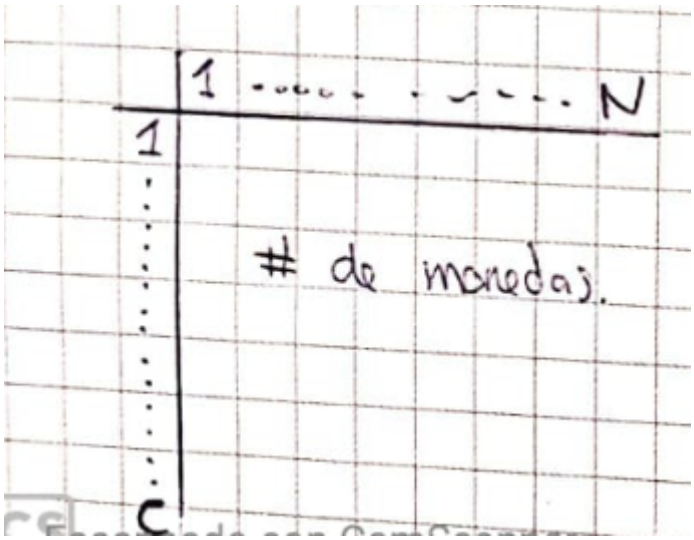
3. Crear una clase que implemente la interfaz CoinChangeCalculator implementando directamente la ecuación de recurrencia como una función recursiva.

   Realizado en Proyecto java adjunto.

4. Crear una clase que implemente la interfaz CoinChangeCalculator implementando un algoritmo voraz que escoja en cada paso la moneda con mayor denominación. Calcular la complejidad temporal de este algoritmo.

   Realizado en Proyecto java adjunto.

5. Dibujar un grafo de necesidades de acuerdo con la ecuación de recurrencia planteada en el segundo punto



6. De acuerdo con el grafo de necesidades, crear una clase que implemente la interfaz CoinChangeCalculator implementando un algoritmo de programación dinámica para resolver el problema. Calcular la complejidad temporal de este algoritmo.

   Realizado en Proyecto java adjunto.

7. Utilizar el programa disponible en la clase ExampleCoinChange para probar los algoritmos. Probar los diferentes algoritmos con valores totales 1000, 100000 y un millon y con 3 distintos conjuntos de denominaciones y generar una tabla con el tiempo que necesitó cada algoritmo

Greedy

Denominaciones\cantidad	**1,000**	**100,000**	**1,000,000**
1,2,5,10,25,50,100	0	0	0
1,3,12,36,67,259	0	0	0
1,6,13,47	0	4	0

Recursive

Denominaciones\cantidad	1,000	100,000	1,000,000
1,2,5,10,25,50,100	0	0	0
1,3,12,36,67,259	0	0	0
1,6,13,47	0	4	Error

DynamicProgramming

Denominaciones\cantidad	1,000	100,000	1,000,000
1,2,5,10,25,50,100	0	4	78
1,3,12,36,67,259	0	8	62
1,6,13,47	0	0	64

8. Describir un valor y un conjunto de denominaciones en el que el algoritmo voraz no encuentra la solución óptima.

   El algoritmo voraz no encuentra la solución óptima en conjunto como:

```
Coin Change Algorithm: Greedy
Coin    Number
1       2
5       1
10      0
12      5
Total coins:    8
Total value:    67
Total time spent (milliseconds): 0
```

```
Coin Change Algorithm: DynamicProgramming
Coin    Number
1       0
5       1
10      5
12      1
Total coins:    7
Total value:    67
Total time spent (milliseconds): 0
```

Parte 2: Otros problemas de programación dinámica

Para los siguientes problemas realice los siguientes pasos:

a. Formalizar el problema describiendo sus entradas, salidas, precondición y postcondición

b. Definir una función con la que se pueda representar el valor a optimizar en el problema

c. Definir una ecuación de recurrencia para calcular dicha función que exprese la solución en términos de soluciones a subproblemas relacionados

d. Dibujar el grafo de necesidades relacionado con la ecuación

e. Diseñar un algoritmo de programación dinámica en GCL que permita obtener cualquier valor de la ecuación de recurrencia

Nota: *No es obligatorio (aunque si es recomendable) desarrollar una implementación del algoritmo.*

1. Dado un arreglo a de números naturales y un número total T, decidir si existe un conjunto C de índices del arreglo tal que:

$(+ i \mid i \in C : a[i]) = T$

Ejemplo de entrada: a=[15,28,3,12,12] y T = 30

Desarrollo:

a. Entradas, salidas, precondición y postcondición

Entrada/Salida	Nombre	Tipo
E	arregloNaturales	Array [0:N] nat
E	T	nat
S	existe	boolean

Precondición:

$(N>0) \wedge (\forall k \mid 0<=k<=N: arregloNaturales[k]<=T)$

Postcondición:

$(existe==true \wedge (+ i \mid i \in C : aregloNaturales[i]) = T) \ V \ (existe==false)$

b. Función con la que se pueda representar el valor a optimizar

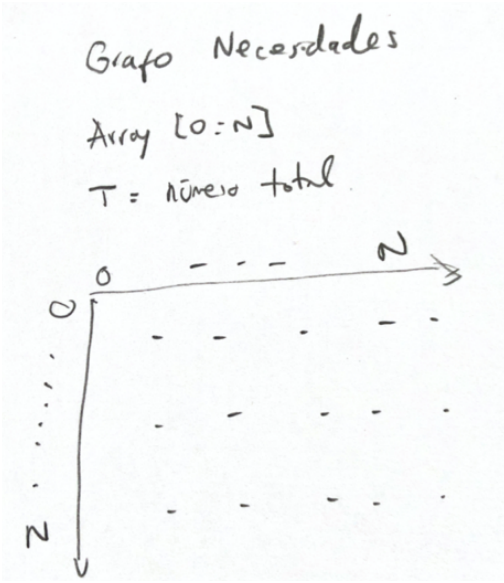
existeConjuntoC(N,T)

c. Ecuación de recurrencia que exprese la solución en términos de soluciones a subproblemas relacionados

existeConjuntoC(i,j) =

Existe==false si i=0
Existe==true si arregloNaturales[i]=T si arregloNaturales[i] =T
max(existeConjuntoC(i-1,j), existeConjuntoC(i,j-1)) si arregloNaturales[i] =!T

d. Grafo de necesidades



e. Algoritmo de programación dinámica en GCL

fun existeConjuntoC(arregloNaturales: array[o][N] of int, T:int) ret existe:boolean

var C : matrix [0,N][0,N]

var i, j:=0,0;

do i<=n → if i=0→existe:=false

[] arregloNaturales[i]=T→existe:=true

[] arregloNaturales[i]!=T → max(existeConjuntoC(i-1,j), existeConjuntoC(i,j-1))

fi

if j<T →j:=j=1;

[] j=T→ i,j:=i+1,0;

fi

od

return existe;

2. Dada una matriz (no necesariamente cuadrada) de unos y ceros, encontrar la cantidad de filas y columnas de la submatriz cuadrada más grande, tal que todos los elementos de dicha submatriz sean iguales a 1.

Desarrollo:

a. Entradas, salidas, precondición y postcondición

Entrada/Salida	Nombre	Tipo
E	matriz	Array[n][m] nat (¿?)
S	numeroFilas	Nat
S	numeroColumnas	nat

Precondición:

$(n>0) \wedge (m>0) \wedge (\forall k \mid 0<=k<=n \wedge 0<=l<=m : matriz[k][l]=1 \wedge matriz[k][l]=0)$

Postcondición:

$(\forall k \mid 0<=k<=numeroFilas \wedge 0<=l<=numeroColumnas : matriz[k][l]=1)$

b. Función con la que se pueda representar el valor a optimizar

Submatriz(n,m)

c. Ecuación de recurrencia que exprese la solución en términos de soluciones a subproblemas relacionados

Submatriz(i,j) =

0 si i=0
Submatriz(i-1,j-1)+1 si sumbatriz[i-1][j-1]=1
max(submatriz(i-1,j), submatriz(i,j-1)) si submatriz[i][j] =!1

d. Grafo de necesidades

d. Algoritmo de programación dinámica en GCL

```
fun submatriz(matriz: array[m][n] of  int) ret numeroFilas:int , numeroColumnas:int

var submatriz : matrix [0,n].[0,m]

var i, j:=0,0;

do i<=n → if i=0 → 0
        [] sumbatriz[i-1][j-1]=1→ Submatriz(i-1,j-1)+1
        []  submatriz[i ][j] =!1→ max(submatriz(i-1,j), submatriz(i,j-1))

        fi

        if  j<m→j:=j+1;

        [] j=m→i,j:=i+1,0;

        fi

od
```

respuesta := submatriz[numeroFilas,NumeroColumnas]

return respuesta;

3. Desarrollar un programa que cuente la cantidad de números de N dígitos en base 4 que no tengan ceros adyacentes. Por ejemplo, para N=10 un número válido sería 3011203320 mientras que uno inválido sería 2113002021

Desarrollo:

a. Entradas, salidas, precondición y postcondición

Entrada/Salida	Nombre	Tipo
E	N	nat
S	cantidadNumeros	nat

Precondición:

(N>0)

Postcondición:

(∀k | 1<=k<=N: matriz[k]=0 → matriz[k-1]!=0) ∧ (+ 1 | i ∈ C : matriz[i]) = cantidadNumeros)

(la sumatoria de los que cumplen la condición es igual a la cantidadNumeros)

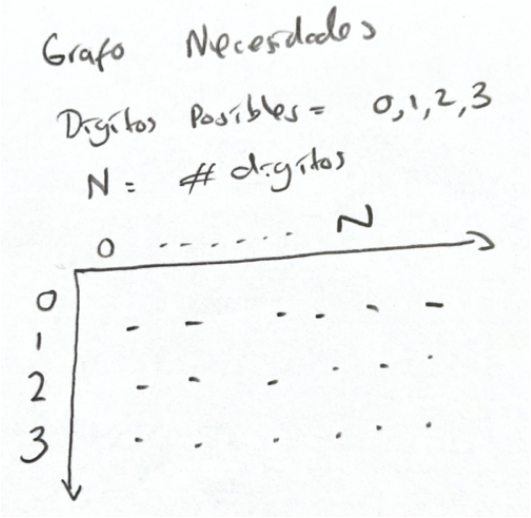
b. Función con la que se pueda representar el valor a optimizar

cantidadSinCerosAdyacentes(4,N)

c. Ecuación de recurrencia que exprese la solución en términos de soluciones a subproblemas relacionados

cantidadSinCerosAdyacentes(i,j)= {
0 si i=0
cantidadSinCerosAdyacentes (i-1,j)+1 si matriz[k]=0 → matriz[k-1]!=0
max(cantidadSinCerosAdyacentes (i-1,j) , cantidadSinCerosAdyacentes (i,j-1)) si si matriz[k]=0 → matriz[k-1]=0

d. Grafo de necesidades



e. Algoritmo de programación dinámica en GCL

fun cantidadSinCerosAdyacentes(N: int) ret cantidadNumeros: int

var matriz : matrix [0,4].[0,N]

var i, j:=0,0;

do i<=n → if → 0 si i=0
[] (matriz[k]=0 → matriz[k-1]!=0) → cantidadSinCerosAdyacentes (i-1,j)+1
[] (matriz[k]=0 → matriz[k-1]=0) → max(cantidadSinCerosAdyacentes (i-1,j) , cantidadSinCerosAdyacentes (i,j-1))
fi

if j<N→j:=j+1;

[] j=N→i,j:=i+1,0;

fi

od

cantidadNumeros:= matrix[4][N] .size

return cantidadNumeros;