Tarea 4

Tony Santiago Montes Buitrago – 202014562

Juan Carlos Marin Morales – 202013973

Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes

19 de octubre de 2021

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 $d_1, d_2, \dots d_n$, determine la cantidad mínima de monedas $m_1, m_2, \dots, m_n$, tal que la suma total de dinero sea igual a P.

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

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

Entradas:

Variable	Tipo	Descripción
c	array of nat [0,N)	Arreglo de números naturales que representa la denominación de todas las posibles monedas que se tienen.
P	nat	Cantidad total de dinero.

Salidas:

Variable	Tipo	Descripción
m	Array of nat [0,N)	Un arreglo de números naturales que representa la cantidad de monedas de cada denominación para la solución optima

b) 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.

$m(i,j)$: *Funcion que calcula el minimo de monedas requeridas para hacer la cantidad solicitada*

i: *i primeras denominaciones de monedas*

j: *cantidad de dinero*

$$m(i,j)\begin{cases} \infty \ si\ i = 0 \wedge j > 0 \\ 0\ si\ j = 0 \\ m(i-1,j)\ si\ j < c_i \\ \min(1 + m(i, j - c_i), m(i-1,j)) \end{cases}$$

Esta ecuación calcula la ecuación diseñada debido a que para cada denominacion de moneda toma la opción entre incluir una moneda mas de esa denominación o excluirla, y así tomar la minima de monedas usadas entre las dos decisiones.

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

VER EN CODIGO

d) 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.

```
4
5  public class CoinChangeGreedy implements CoinChangeAlgorithm{
6
7      @Override
8      public int[] calculateOptimalChange(int totalValue, int[] denominations) {
9          Arrays.sort(denominations);
10         int[] ans = new int[denominations.length];
11         int i = denominations.length -1;
12         while (i >= 0) {
13             while(totalValue >= denominations[i]) {
14                 totalValue -= denominations[i];
15                 ans[i]++;
16             }
17             i--;
18         }
19         return ans;
20     }
21
22 }
23
```

Calculo de complejidad:

Siendo n la cantidad de monedas.

Operaciones:

Arrays.sort: nlog(n). (Suponiendo que se realiza con mergesort)

Constructor: c1

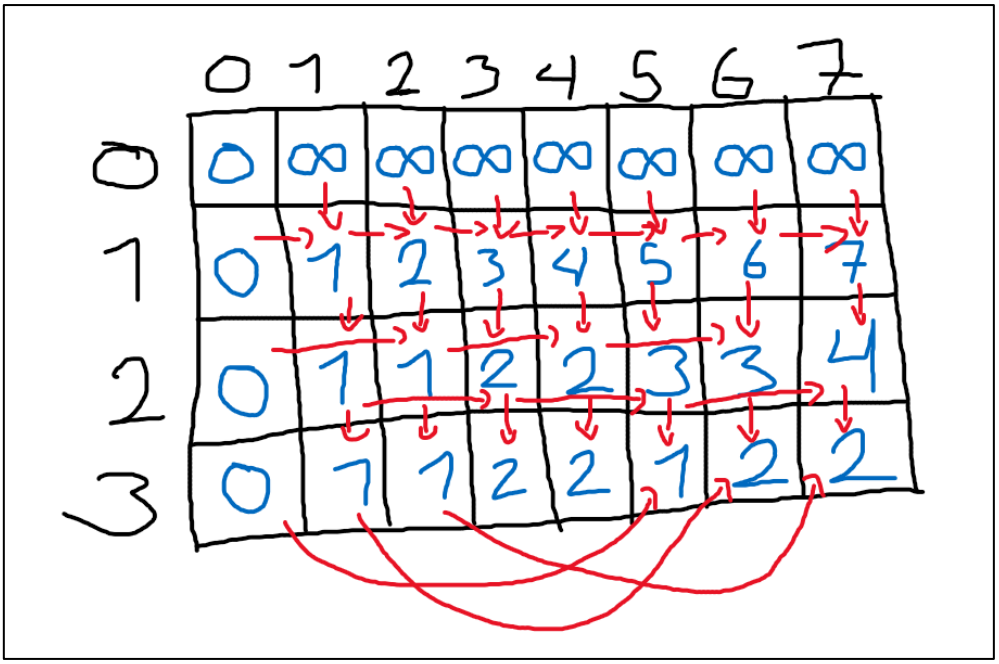
Asignacion: 2c2 + n*3c2

$f(n) = c_1 + 2c_2 + 3nc_2$

$$O\big(f(n)\big) = O(n)$$

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

Grafo de necesidades para las entradas c = [1,2,5] P = 7



f) 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.

```
public class CoinChangeDynamic implements CoinChangeAlgorithm {

    @Override
    public int[] calculateOptimalChange(int totalValue, int[] denominations) {
        int[][] m = new int[denominations.length+1][totalValue+1];
        for (int i = 0; i <= denominations.length; i++) {
            for(int j = 0; j <= totalValue; j++) {
                if (j == 0) { m[i][j] = 0;}
                else if (i == 0 && j > 0) {m[i][j]= Integer.MAX_VALUE;}
                else if (denominations[i-1] > j) {m[i][j] = m[i-1][j];}
                else {m[i][j] = Math.min(1+m[i][j-denominations[i-1]], m[i-1][j]);}
            }
        }
        int[] res = new int[denominations.length];
        int i = denominations.length;
        int j = totalValue;
        while (i>0 && j >0) {
            if  (denominations[i-1]<=j && m[i][j] == m[i][j-denominations[i-1]]+1) {
                res[i]++;
                j-= denominations[i-1];
            }
            else if (denominations[i-1]>j && m[i][j] == m[i-1][j]) {
                i--;
                }
        }
        return res;
    }

}
```

Este algoritmo tiene un orden de complejidad de (n*P) siendo n el numero de monedas y siendo P la cantidad de dinero.

Parte 2. Otros problemas de programación dinámica

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 o a contar en el problema o que represente la respuesta en el caso de problemas de si o no.
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 que permita obtener cualquier valor de la ecuación de recurrencia. Puede escribir el algoritmo en el lenguaje de su elección.

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 $(+i|i \in C: a[i]) = T$.
Ejemplo de entrada: $a = [15,28,3,12,12]$ y $T = 30$.

a) Entradas:

Variable	Tipo	Descripción
a	array of nat [0,N)	Arreglo de números naturales
T	nat	Número el cual se busca un conjunto cuya suma sea igual a este.

Salidas:

Variable	Tipo	Descripción
b	bool	Existe un subconjunto de a tal que su suma sea igual a T

b) Función de Optimización:
$m(i,j)$: *Subconjunto de los i primeros elementos cuya suma es o no es el valor de j.*

c)

$$m(i,j) = \begin{cases} True\ si\ j = 0 \\ False\ si\ i = 0 \\ m(i-1, j-a_i) \vee m(i-1,j)\ si\ i,j > 0 \end{cases}$$

d) Grafo de necesidades para

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	True	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
15	1	True	False	False	False	False	False	False	False	False	False	False	False	False	False	False	True
28	2	True	False	False	False	False	False	False	False	False	False	False	False	False	False	False	True
3	3	True	False	False	True	False	False	False	False	False	False	False	False	False	False	False	True
12	4	True	False	False	True	False	False	False	False	False	False	False	False	True	False	False	True
12	5	True	False	False	True	False	False	False	False	False	False	False	False	True	False	False	True

Ilustración 1. Grafo de necesidades, punto 2.1 (a = [15,28,3,12,12], T= 15)

e) Implementación Programación Dinámica en **Python**:

```python
def subConjuntoIgualASuma(a: list, T: int)->bool:
    m = []
    for i in range(len(a)+1):
        row = []
        for i in range(T+1):
            row.append(False)
        m.append(row)

    for i in range(len(m)):
        for j in range(len(m[i])):
            if j == 0:
                m[i][j] = True
            elif i == 0:
                m[i][j] = False
            else:
                if j-a[i-1] < 0:
                    m[i][j] = m[i-1][j]
                else:
                    m[i][j] = m[i-1][j-a[i-1]] or m[i-1][j]
    return m[len(m)][len(m)[0]]
```

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.

a) Entradas:

Variable	Tipo	Descripción
x	array[0,N) of array[0,M) of int	Matriz de unos y ceros

Salidas:

Variable	Tipo	Descripción
z	int	Número de filas y columnas de la matriz más grande dentro de x, que solo contenga unos

Precondición:

$$\boldsymbol{Q} \equiv \left(\forall\langle \boldsymbol{i},\boldsymbol{j}\rangle \in \mathbb{Z}^{\boldsymbol{2}} \middle| (\boldsymbol{0} \leq \boldsymbol{i} < \boldsymbol{N}) \wedge (\boldsymbol{0} \leq \boldsymbol{j} < \boldsymbol{M}) : \boldsymbol{x}[\boldsymbol{i}][\boldsymbol{j}] \in (\boldsymbol{0},\boldsymbol{1})\right)$$

Postcondición:

Sean…

$$B(k,i,l,j) \equiv (\forall\langle m,n\rangle \in \mathbb{Z}^2 | (k \leq m \leq i) \wedge (l \leq n \leq j) : x[m][n] = 1) \Rightarrow (z \geq i-k)$$

$$A(i,j) \equiv \left(\forall\langle k,l\rangle \in \mathbb{Z}^2 | (0 \leq k < i) \wedge (0 \leq l < j) \wedge (i-k = j-l) : B(k,i,l,j)\right)$$

Se tiene la postcondición:

$$\boldsymbol{R \equiv \left(\forall \langle i,j \rangle \in \mathbb{Z}^2 | (0 \leq i < N) \wedge (0 \leq j < M) : A(i,j)\right)}$$

En palabras: Para toda submatriz cuadrada de x, si todos los números de la submatriz son 1, entonces z es mayor o igual que el número de filas de la submatriz.

b) Función de Optimización:

$$f(i,j)$$

$i =$ número de filas que se analizan, desde 0 ($0 \leq i \leq N$)

$j =$ número de columnas que se analizan, desde 0 ($0 \leq j \leq M$)

c)

$$f(i,j) = \begin{cases} 0\,, & x[i][j] = 0 \\ \begin{pmatrix} \left(\max\left(f(i-1,j), f(i,j-1), f(i-1,j-1)\right)\,, & d.l.c. \\ f(i-1,j)+1\,, & f(i-1,j) = f(i,j-1) = f(i-1,j-1) \end{pmatrix}, & \\ & x[i][j] = x[i][j-1] = x[i-1][j] = x[i-1][j-1] = 1 \\ 1\,, & d.l.c. \end{cases}$$

NOTA: La ecuación de recurrencia puede ser un poco compleja de leer y entender, para esto se recomienda ver la versión de código (literal **e)**) para comprenderla mejor.

NOTA: *d.l.c.* son las siglas de "de lo contrario".

d)

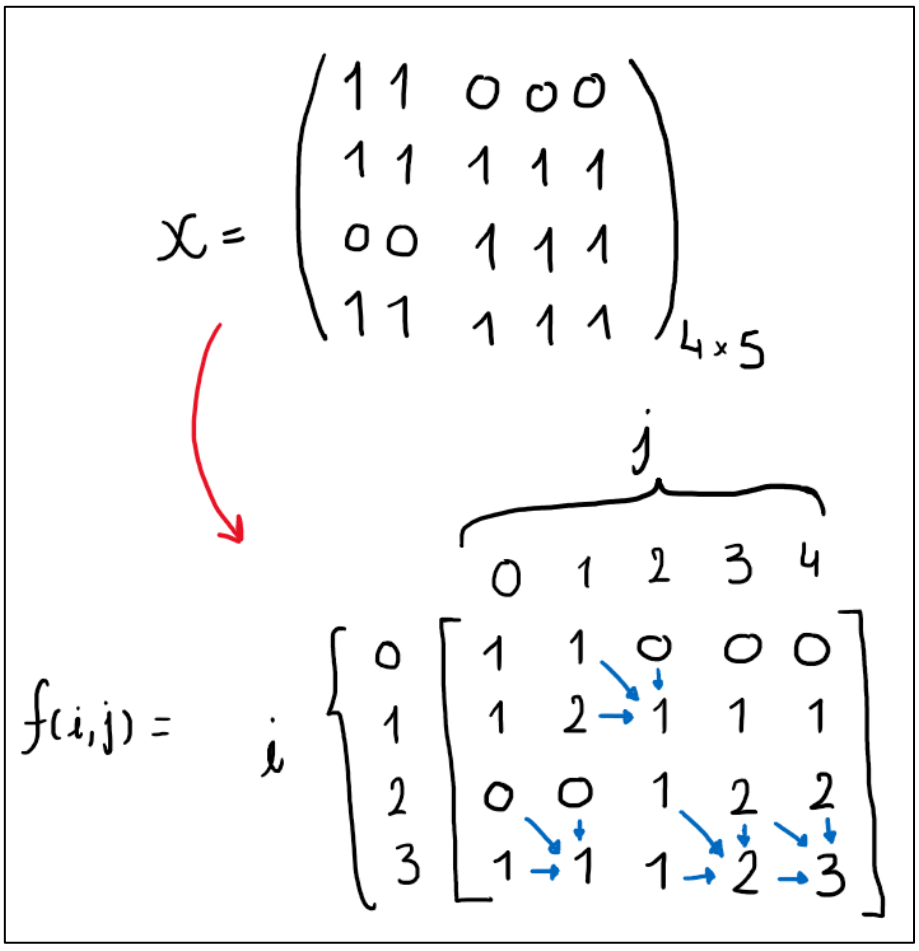


Ilustración 22. Grafo de necesidades, punto 2.2

e) Implementación Programación Dinámica en **Python**:

```python
def matrixOnes(x:list):

    N,M=len(x),len(x[0])
    mtx = [[x[i][j]
            for j in range(M)]
            for i in range(N)] #crea matrix de ceros
    i,j,run_i,tmp = 0,1,True,1
    while True:
        i,j = (i+1,j) if run_i else (i,j+1)
        if i>=N or j>=M: break

        if x[i][j]==0:
            r = 0
        else:
            if x[i-1][j] == x[i][j-1] == x[i-1][j-1] == 1:
                if mtx[i-1][j] == mtx[i][j-1] == mtx[i-1][j-1]:
                    r = mtx[i-1][j] + 1
                else:
                    r = max(mtx[i-1][j], mtx[i][j-1], mtx[i-1][j-1])
            else:
                r = x[i][j]
        mtx[i][j] = r

        if i==N-1:
            run_i,i,tmp = False,tmp,j+1
        if j==M-1:
            run_i,j,tmp = True,tmp,i+1

    return max([max(k) for k in mtx])
```

NOTA: Nótese que, a diferencia de la mayoría de los ejercicios de programación dinámica, la respuesta final no queda en el último elemento de la matriz; para hallar la respuesta se debe hallar el máximo valor que haya en la matriz.

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

a) Entradas:

Variable	Tipo	Descripción
N	nat	Número de dígitos en base 4 que debe tener cada cadena a evaluar

Salidas:

Variable	Tipo	Descripción

r	int	Cantidad de números de N dígitos en base 4 que no tienen 0 adyacentes en ninguna parte

Precondición:

$$\boldsymbol{Q \equiv true}$$

Postcondición:

Sea…

ξ_4: Conjunto de todos los números en base 4; donde cada número está representado como una lista de números donde cada elemento es un dígito (ej: $[1,3,0] \in \xi_4$)

Se tiene la postcondición:

$$\boldsymbol{R \equiv \left(r = \left(\Sigma\, n \in \xi_4 \mid (|n| = N) \wedge \left(\forall i \mid 0 \leq i < N-1 : (n[i] = 0) \Rightarrow (n[i+1] \neq 0)\right) : 1\right)\right)}$$

En palabras: r es el conteo de números en base 4 que cumplen 2 condiciones:

1. Tiene N dígitos
2. Si un dígito es 0 entonces el siguiente no es 0.

b) Función de Optimización:

$$f(i)$$

$i =$ Número de dígitos de los números en base 4 $(0 \leq i \leq N)$

c)

$$f(i)\begin{cases} 1, & i = 0 \\ 4, & i = 1 \\ 3 \cdot \left(f(i-1) - f(i-2)\right), & i > 1 \end{cases}$$

d)

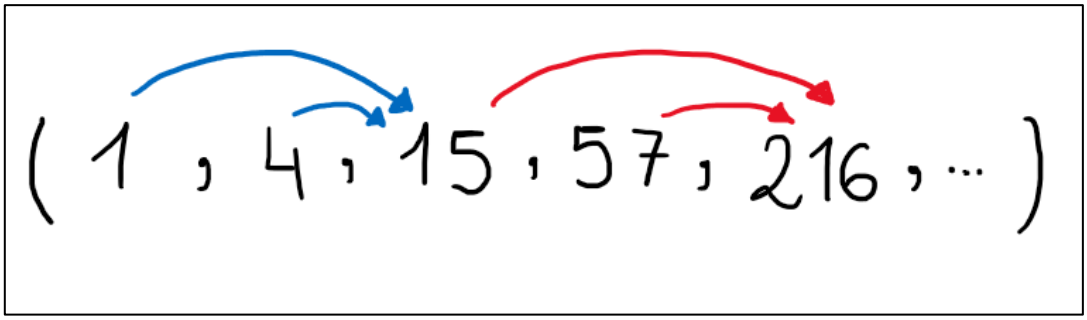


Ilustración 33. Grafo de necesidades, punto 2.3

e) Implementación Programación Dinámica en **Python**:

```python
def base4nums(N:int):

    lst = []
    i = -1
    while (i := i+1) <= N:
        if i==0:
            r = 1
        elif i==1:
            r = 4
        else:
            r = 3*(lst[i-1]+lst[i-2])
```

```
        lst.append(r)
        i+=1
    return lst[N]
```

4) El profesor de un colegio decidió repartirle dulces a sus N estudiantes de acuerdo con la nota que sacaron en un examen. Para que ninguno se sienta mal, le va a repartir por lo menos un dulce a cada estudiante. Para facilitar este proceso, decide sentar los estudiantes en una sola fila. Sin embargo, como los estudiantes hablan entre ellos, cada estudiante sabe su propia nota y la de sus dos vecinos. De esta forma, un estudiante que haya sacado más nota que un vecino espera recibir más dulces que dicho vecino. El profesor necesita entonces un programa que le ayude a calcular cuántos dulces necesitaría repartir como mínimo para cumplir estas condiciones.
Nota: Si dos estudiantes que sean vecinos sacaron la misma nota, deben recibir la misma cantidad de dulces.

a) Entradas:

Variable	Tipo	Descripción
e	array[0,N) of float	Lista de notas de los estudiantes en el orden en el que están la fila

Salidas:

Variable	Tipo	Descripción
sum	int	Número mínimo de dulces necesarios para repartir, cumpliendo las condiciones del profesor

Precondición:

$$\boldsymbol{Q \equiv (\forall i | 0 \leq i < N : 0 \leq e[i] \leq 5)}$$

Postcondición:

Sea…

$lst =$ Lista con el número de dulces que se le dará a cada estudiante (sobre la cual se calcula sum)

$$\boldsymbol{R \equiv \Big(\forall i | 1 \leq i < N - 1 : \big((e[i] > e[i+1]) \Rightarrow (lst[i] > lst[i+1])\big) \wedge \big((e[i] > e[i-1]) \Rightarrow (lst[i] > lst[i-1])\big)\Big)}$$

b) Función de Optimización:

$$d(i)$$

$i =$ Número de estudiantes analizados desde 0 hasta i $(0 \leq i < N)$

Funciones Auxiliares:

$$d_1(i)$$

$i =$ Número de estudiantes analizados desde 0 hasta i $(0 \leq i < N)$, teniendo en cuenta solo al estudiante vecino de la izquierda.

$$d_2(i)$$

i = Número de estudiantes analizados desde i hasta N-1 $(0 \leq i < N)$, teniendo en cuenta solo al estudiante vecino de la derecha.

c)

$$d(i) = \max\big(d_1(i), d_2(i)\big)$$

NOTA: d_1 y d_2 ya calculan los mínimos valores necesarios; sin embargo, para que se cumplan correctamente las condiciones, d debe hallar el máximo de estas dos funciones.

$$d_1(i) = \begin{cases} 2\,, & \big((i = 0) \wedge (e[0] > e[1])\big) \\ 1\,, & \big((i = 0) \wedge (e[0] \leq e[1])\big) \vee \big((i > 0) \wedge (e[i] < e[i-1])\big) \\ d_1(i-1) + 1\,, & (i > 0) \wedge (e[i] > e[i-1]) \\ d_1(i-1)\,, & (i > 0) \wedge (e[i] = e[i-1]) \end{cases}$$

$$d_2(i) = \begin{cases} 2\,, & \big((i = N-1) \wedge (e[N-1] > e[N-2])\big) \\ 1\,, & \big((i = N-1) \wedge (e[N-1] \leq e[N-2])\big) \vee \big((i > N-1) \wedge (e[i] < e[i+1])\big) \\ d_2(i+1) + 1\,, & (i < N-1) \wedge (e[i] > e[i+1]) \\ d_2(i+1)\,, & (i < N-1) \wedge (e[i] = e[i+1]) \end{cases}$$

d)

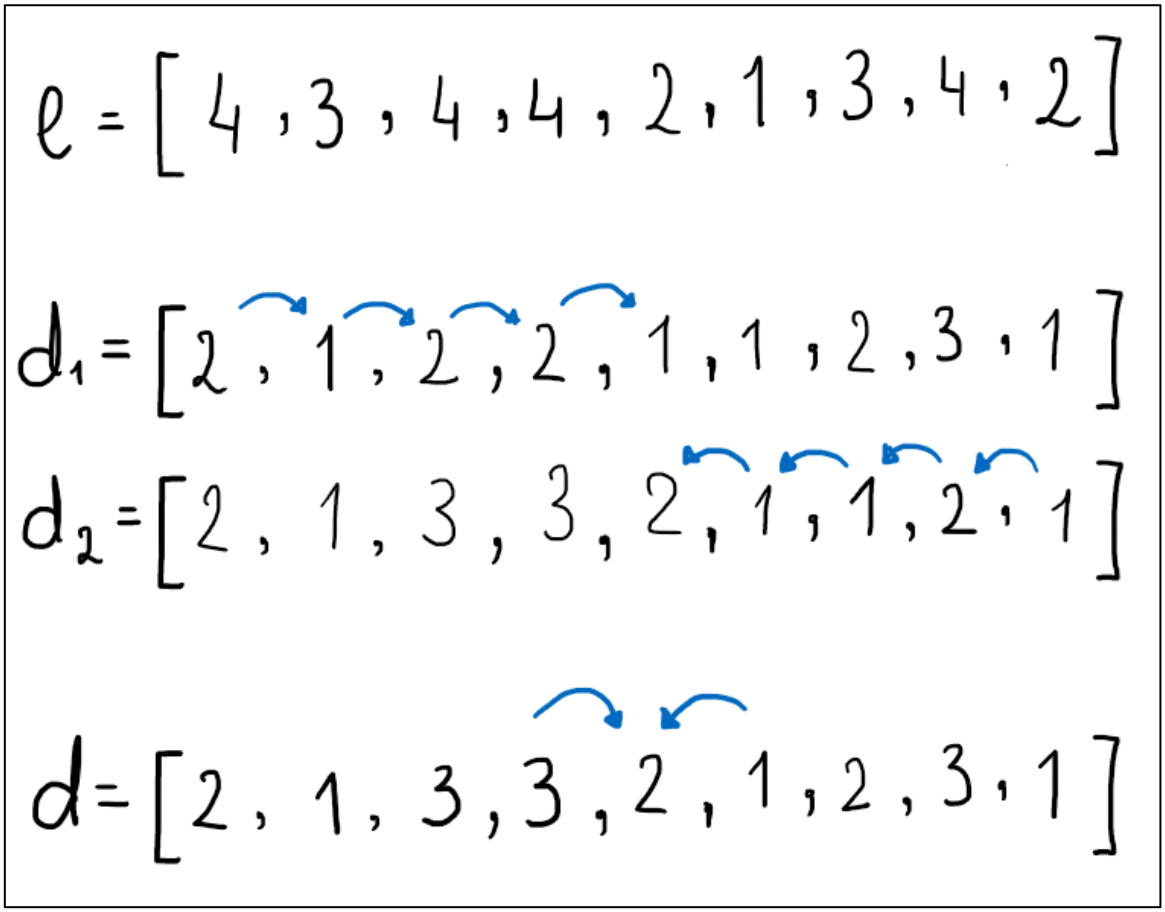


Ilustración 44. Grafo de necesidades, punto 3.4

e) Implementación Programación Dinámica **Python**:

```python
def _lC1(e:list, lstleft:list, i:int):
    if i==0:
        r = (2 if e[0]>e[1] else 1)
    else:
        if e[i]<e[i-1]:
            r = 1
        elif e[i]>e[i-1]:
            r = lstleft[i-1] + 1
        else:
            r = lstleft[i-1]
    lstleft.append(r)

def _lC2(e:list, lstright:list, i:int):
    if i==len(e)-1:
        r = (2 if e[len(e)-1]>e[len(e)-2] else 1)
    else:
        if e[i]<e[i+1]:
            r = 1
        elif e[i]>e[i+1]:
            r = lstright[i+1] + 1
        else:
            r = lstright[i+1]
    lstright[i] = r

def lessCandies(e:list):
    lstleft = []
    lstright = [0 for i in range(len(e))]
    i = -1
    while (i := i+1) < len(e):
        _lC1(e,lstleft,i)
    i = len(e)
    while (i := i-1) >= 0:
        _lC2(e,lstright,i)
    i,sum = -1,0
    while (i := i+1) < len(e):
        sum += max(lstleft[i], lstright[i])
    return sum
```

5) En un juego de video el protagonista tiene que atravesar un recorrido lineal de N metros. En cada metro pueden ocurrir una de dos cosas:

1. Hay un trampolín que le permite saltar una cantidad de metros entre 2 y K hacia adelante. El protagonista puede decidir si usa el trampolín para saltar o si simplemente camina un metro hacia adelante.
2. Hay un abismo en el que, si cae, pierde el juego.

Se debe desarrollar un programa que determine si existe algúna forma de llegar al final del recorrido. Se debe llegar exactamente al metro N porque después de este metro hay un abismo.

a) Entradas:

Variable	Tipo	Descripción
c	array [0,N) of int	Lista de casillas en cada metro del juego que indican la cantidad de casillas que puede saltar (-1 si es un abismo)

Salidas:

Variable	Tipo	Descripción
z	bool	True si existe alguna forma de completar el juego, False de lo contrario

Precondición:

$$\boldsymbol{Q \equiv (\forall i | 0 \leq i < N : -1 \leq c[i])}$$

Postcondición:

Sea…

b: Lista de booleanos que indican en cada casilla el resultado de f(i)

$$\boldsymbol{R \equiv \left(z \Rightarrow \left(\forall i | 0 \leq i < N : b[i] \Rightarrow (b[i+1] \vee b[i + c[i]])\right)\right)}$$

b)

$$f(i)$$

$i =$ Número de casillas que se analizan desde i hasta N $(0 \leq i < N)$

c)

$$f(i) = \begin{cases} false\,, & (c[i] = -1) \vee (i + c[i] \geq N) \\ true\,, & (i = N-1) \wedge (c[i] = 0) \\ f(i+1) \vee f(i + c[i])\,, & d.l.c. \end{cases}$$

d)

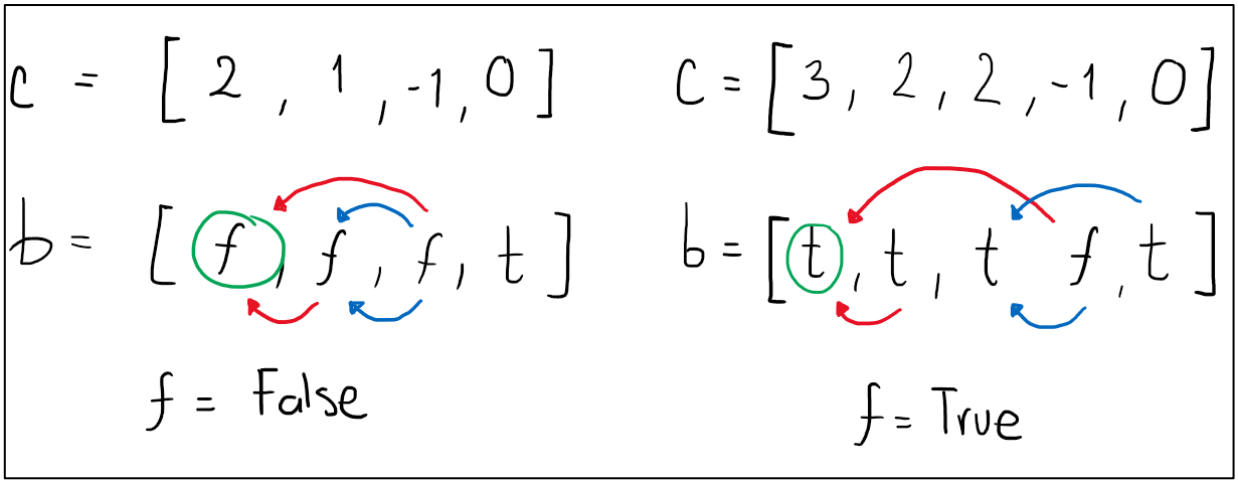


Ilustración 55. Grafo de necesidades, punto 2.5

e)

```python
def Nmeters(c:list):
    b = [None for i in range(len(c))]
    i = len(c)
    while (i := i-1) >= 0:
        if (c[i]==-1) or (i+c[i]>=len(c)):
            r = False
        elif (i==len(c)-1) and (c[i]==0):
            r = True
        else:
            r = b[i+1] or b[i+c[i]]
        b[i] = r
    return b[0]
```