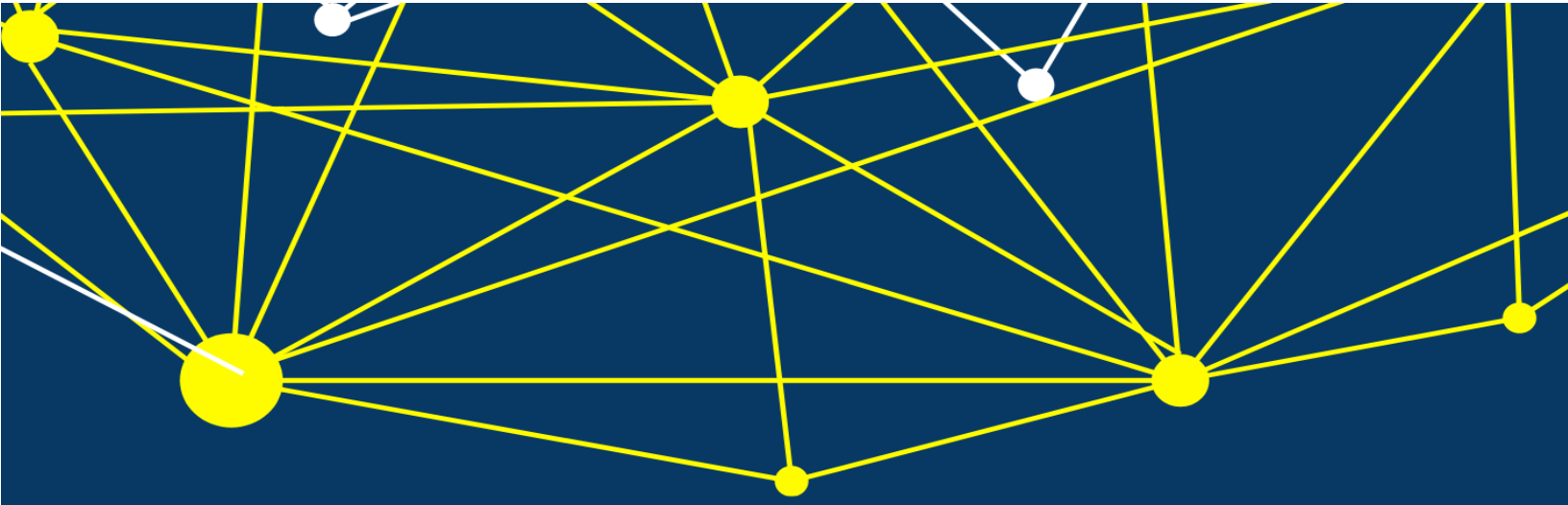

Adrián Borges Cano

Marco González Martínez

Laboratorio: Miércoles 15:00-17:00

Algoritmia

ÍNDICE

Tabla de contenido

PROBLEMA 3

Se tiene un sistema de billetes de distintos valores y ordenados de menor a mayor (por ejemplo 1, 2, 5, 10, 20, 50 y 100 euros), que se representan mediante los valores vi, con $i \in \{1, \dots, N\}$ (en el caso anterior, = 7) de manera que de cada billete se tiene una cantidad finita, mayor o igual a cero, que se guarda en ci (siguiendo con el ejemplo, $c3$ = 6 representaría que hay 6 billetes de 5 euros). Se quiere pagar exactamente una cierta cantidad de dinero D, utilizando para ello la menor cantidad de billetes posible. Se sabe que $D \leq \sum civini$=1 , pero puede que la cantidad exacta D no sea obtenible mediante los billetes disponibles. Diseñar un algoritmo con la metodología de Programación Dinámica que determine, teniendo como datos los valores ci, vi y D :

• si la cantidad D puede devolverse exactamente o no, y

• en caso afirmativo, cuantos billetes de cada tipo forman la descomposición óptima.

Descripción del algoritmo:

Para poder resolver este ejercicio, hemos decidido construir todas las combinaciones posibles con los billetes ofrecidos, hasta llegar a la cantidad pedida. Este proceso se ha realizado iterativamente, de forma que hemos comenzado primero con los billetes de menor valor, y terminado con los de mayor. Cuando se encontraba una combinación posible con los billetes de mayor valor (siempre y cuando esto implique utilizar un menor número de billetes totales, siendo esto una solución posible a partir de los billetes disponibles) la opción encontrada se sustituía por la que se había calculado en un primer momento. De esta forma, al final del proceso, obtenemos una lista donde se muestran las opciones óptimas para cada cantidad requerida desde 1 hasta N, donde devolvemos únicamente el último elemento de esta lista, puesto que es el requerido.

Si la N introducida no es posible ser hallada con el conjunto de billetes dado, lejos de devolver la solución, devolverá un 0 como muestra de que lo que quiere el usuario no es posible.

A modo de cierre, no queríamos olvidar mencionar el uso de algunos métodos externos creados por nosotros mismos por higiene del código.

Casos de prueba:

Caso 1:

- **Cantidad: 11**
- **Valor: 1, 3, 5**
- **Cantidad de monedas: 15, 5, 5**

Caso 2:

- **Cantidad: 11**
- **Valor: 1, 3, 5**

- **Cantidad de monedas: 3, 5, 2**

Caso 3:

- **Cantidad: 201**
- **Valor: 1, 20, 50, 100**
- **Cantidad de monedas: 1, 10, 5, 1**

```
print(billetes(11,[1,3,5],[15,5,5]))
print(billetes(11,[1,3,5],[3,5,2]))
print(billetes(201,[1,20,50,100],[1,10,5,1]))
```

Resultado para el Caso 1 → [[1, 1], [3, 0], [5, 2]]

Resultado para el Caso 2 → [[1, 1], [3, 0], [5, 2]]

Resultado para el Caso 3 →[[1, 1], [20, 0], [50, 2], [100, 1]]

La primera posición de la pareja, indica el valor del billete, la segunda posición indica la cantidad del mismo.

Para analizar el paso a paso de los casos de prueba, explicaremos solamente uno de ello, el Caso 1, ya que el resto tendrán un procedimiento muy parecido:

1. Lo primero que hacemos es crear la tabla que contendrá todas las soluciones óptimas del ejercicio, siendo inicializados todos sus elementos con valor 0.
2. Más tarde, comprobamos si un único billete del que se está evaluando puede ser la solución a un elemento de la tabla. Como en este caso comenzamos con el billete de 1, valdrá para el primer elemento de la tabla.
3. A continuación, introducimos iterativamente las combinaciones posibles (siempre que el número de billetes dados como parámetro de entrada lo permitan) con el primer tipo de billetes; en este caso, todas las combinaciones posibles con billetes de 1, hasta N=11, puesto que el número de billetes dado es 15>11.
4. Después, se realiza lo mismo con el siguiente tipo de billetes, en este caso con los billetes de 3. Además, ahora se comprueba si la nueva combinación encontrada es mejor que la opción anterior, y en caso afirmativo, se sustituye por el nuevo valor.
5. Finalmente, una vez hemos realizado todas las iteraciones, terminando con los billetes de 5, devolvemos el último elemento de la lista, puesto que es la opción pedida.

A continuación, se adjunta una tabla gráfica de la estructura de datos del programa:

	1	2	3	4	5	6	7	8	9	10	11
Uno	1 Uno	2 Uno	3 Uno	4 Uno	5 Uno	6 Uno	7 Uno	8 Uno	9 Uno	10 Uno	11 Uno
Tres	1 Uno	2 Uno	1 Tres	1 Tres, 1 Uno	1 Tres, 2 Uno	2 Tres	2 Tres, 1 Uno	2 Tres, 2 Uno	3 Tres	3 Tres, 1 Uno	3 Tres, 2 Uno
Cinco	1 Uno	2 Uno	1 Tres	1 Tres, 1 Uno	1 Cinco	1 Cinco, 1 Uno	1 Cinco, 2 Uno	1 Cinco, 1 Tres	1 Cinco, 1 Tres, 1 Uno	2 Cinco	2 Cinco, 1 Uno

Código del algoritmo:

```
def combinacion(a,b):
    # Si uno de los candidatos a combinar no ofrece solución, la combinación tampoco.
    if a==0 or b==0:
        return 0
    c = []
    for i in range(len(a)):
        c.append([a[i][0],a[i][1]+b[i][1]])
    return c

def num_monedas(candidato):
    tot= 0
    for i in range(len(candidato)):
        tot+=candidato[i][1]
    return tot

def mejor_candidato(candidatos,c):
    # Descartar candidatos no válidos, aquellos que no ofrezcan solución (0s) o aquellos que usen más
monedas de las que se dispone
    i = 0
    while i < len(candidatos):
        valido = True
        if candidatos[i]==0:
            valido=False
        else:
            for j in range(len(c)):
                if candidatos[i][j][1]>c[j]:
                    valido=False
        if not(valido):
            del(candidatos[i])
        else:
            i+=1
    #Comparar los dos primeros candidatos hasta que sólo quede el que utilice menos monedas
    while len(candidatos)>1:
        if num_monedas(candidatos[1])>=num_monedas(candidatos[0]):
```

```
            del(candidatos[1])
        else:
            del(candidatos[0])
    if len(candidatos)>0:
        return candidatos[0]
    else:
        return 0

def billetes(d, v, c):
    """
    int, list(int), list(int) --> bool, string
    """
    cantidades_a_devolver = list(range(1,d+1)) # Se crea una lista de aquellas cantidades para las que se va a buscar una solución al problema

    # Inicialización de la tabla de la solución
    sol = []
    for i in range(len(v)):
        sol_fil = []
        for j in range(len(cantidades_a_devolver)):
            sol_fil.append(0)
        sol.append(sol_fil)

    for i in range(len(v)):
        for j in range(len(cantidades_a_devolver)):
            if v[i]==cantidades_a_devolver[j]:
                # Usar 1 billete del valor de billetes nuevo que se está evaluando, siempre será la solución óptima para devolver esa misma cantidad
                # Sea un ejemplo: devolver 5 Euros con billetes de 5 Euros -> Se usa 1 Billete de 5 Euros
                celda=[]
                for val in v:
                    celda.append([val,0])
                celda[i]=[v[i],1]
                sol[i][j]=celda
            else:
                # Candidatos es la lista de opciones a elegir para la nueva celda que se está evaluando
```

```
            candidatos = []
            # Añadir a los candidatos la solución para devolver la cantidad requerida con las monedas
anteriores
            candidatos.append(sol[i-1][j])
            # Añadir a los candidatos las combinaciones de soluciones cuya cantidad a devolver sumen la
cantidad requerida
            for k in range(j//2+1):
                candidatos.append(combinacion(sol[i][k],sol[i][j-k-1]))
                sol[i][j]=mejor_candidato(candidatos,c)

    # print(sol) # Descomentar para visualizar la tabla completa
    return sol[-1][-1]
```

La complejidad de este algoritmo es de orden $O(n^5)$. Con la mera observación del algoritmo, y por tanto, de los bucles anidados, es trivial deducir que el programa tendrá este orden de complejidad.

PROBLEMA 6

¡Llega el torneo de EscobaBall, y más salvaje que nunca! Este año, en el Colegio de Magia y Hechicería han decidido que los cuatro equipos (Grifos, Serpientes, Cuervos y Tejones) jueguen en cada partido todos contra todos, y como siempre que ningún partido termine en empate. El torneo acabará cuando un mismo equipo haya ganado un total de N partidos (no necesariamente consecutivos). El aprendiz de mago Javi Potter quiere apostar algo de dinero por su equipo, los Grifos, así que se dirige a la casa de apuestas de los gnomos para ver cuánto le darían si gana su equipo: sus ganancias serían iguales a la cantidad de dinero apostado dividido por la probabilidad de que el equipo gane el campeonato (por ejemplo, si los Grifos tuviesen un 50% de ganar el campeonato y Javi apuesta 10 monedas de oro, sus posibles ganancias serían 10/0'5 = 20 monedas de oro; si tuviesen un 20% de ganar, el beneficio posible sería de 10/0'2 = 50 monedas de oro, mayor recompensa al ser más difícil de conseguir). Para obtener esta probabilidad, la casa de apuestas tiene un Valor de Calidad asignado a cada equipo (que mide la habilidad de los jugadores, su motivación, etc, y que es un valor fijo para el equipo e independiente del partido que esté jugando) de manera que cuanto mayor es el VC de un equipo, más probabilidades tiene de ganar un partido. Por ejemplo, si los cuatro equipos tuviesen igual VC todos tendrían un 25% de ganar un partido. Si tres equipos tuviesen el mismo VC y el cuarto equipo tuviese el doble de esa cantidad, los primeros tendrían un 20% y el último un 40%. Como los partidos no pueden terminar en empate, la suma de las probabilidades siempre es el 100%. Teniendo como datos los Valores de Calidad de los equipos, la cantidad de partidos N que debe ganar un equipo para conseguir ganar el torneo, y el dinero D apostado por Javi Potter, obtener cuáles serían las ganancias si ganasen los Grifos.

Descripción del algoritmo:

Para este problema, hemos decidido afrontarlo planteando una matriz de 4 dimensiones, la cual contiene las probabilidades de que cierta combinación de victorias se dé en algún momento del partido. De las 4 coordenadas de la matriz, la 1ª representa el número de victorias de los Grifos, y el resto de coordenadas, los otros equipos (no es importante conocer el orden de Serpientes, Cuervos y Tejones, dado que son equipos por los que no se apuesta). Inicialmente, los 4 equipos no habrán ganado ningún partido todavía; de forma que la probabilidad de que estos equipos tengan 0 victorias en algún momento es del 100%. Después, el resto de probabilidades se calculan a base de multiplicar las probabilidades de ganar un partido de cada equipo, por los valores que ya figuran en la tabla. Para los casos en los que más de un equipo tengan N victorias, como el torneo habría finalizado en cuanto el primero de estos hubiese llegado a la cantidad de victorias, evidentemente esta opción tiene probabilidad 0.

Finalmente, se suman todas las opciones que dan N victorias al equipo de los Grifos (que les hacen ganar el torneo) lo cual nos dará la probabilidad de que el equipo de los Grifos gane el torneo. Por tanto, las ganancias de Javi Potter se obtendrían a partir de la división entre esta probabilidad.

Casos de prueba:

Primer caso.

- **Número de partidos: 2**
- **Valor de Calidad de cada equipo: 40, 20, 15, 25**
- **Dinero apostado: 10**

Segundo caso

- **Número de partidos: 3**
- **Valor de Calidad de cada equipo: 15, 35, 20, 30**
- **Dinero apostado: 10**

```
print(torneo(2,[40,20,15,25],10))
print(torneo(3,[15,35,20,30],10))
```

Resultado para el Primer Caso → 36.231884057971

Resultado para el Segundo Caso → 1167.439635745992

A continuación, explicaremos el funcionamiento del algoritmo mediante la realización del Primer Caso:

1. Al comienzo del algoritmo se crea la tabla con las probabilidades ya mencionada, donde todas las casillas tienen valor 0.

2. Después se realiza un bucle anidado, utilizado para ir rellenando iterativamente los valores de la tabla hasta que lleguemos a la longitud definida por N. Los datos que se van insertando es la multiplicación de la probabilidad del equipo (en este caso, las probabilidades respectivas de 0'4, 0’2, 0’15 y 0’25), con los adyacentes de la casilla.
3. Cuando esta iteración se ha terminado, se suma en la variable ‘acumulador’ todas las probabilidades que dan la victoria al equipo de los Grifos.
4. Finalmente, se devuelve la cantidad apostada por Javi Potter (10) dividido entre la probabilidad final hallada en el acumulador.

Adjuntamos una tabla para facilitar la comprensión de lo ocurrido en el algoritmo. Las casillas moradas representan los valores que contendrá la matriz, mientras que los valores en amarillo representan los valores que, sumados, dan la probabilidad de que gane el torneo los Grifos:

PG= Probabilidad de Grifos

PS= Probabilidad de Serpientes

PT= Probabilidad de Tejones

PC= Probabilidad de Cuervos

		Grifos														
		0					1					2				
				Cuervos					Cuervos					Cuervos		
				0	1	2			0	1	2			0	1	2
Serpientes	0	Tejones	0	1	PC	PC2	Tejones	0	PG	PG·PC	PG·PC2	Tejones	0	PG2	PG2·PC	0
			1	PT	PT·PC	PT·PC2		1	PG·PT	PG·PT·PC	PG·PT·PC2		1	PG2·PT	PG2·PT·PC	0
			2	PT2	PT2·PC	0		2	PG·PT2	PG·PT2·PC	0		2	0	0	0
				Cuervos					Cuervos					Cuervos		
				0	1	2			0	1	2			0	1	2
	1	Tejones	0	PS	PS·PC	PS·PC2	Tejones	0	PG·PS	PG·PS·PC	PG·PS·PC2	Tejones	0	PG2·PS	PG2·PS·PC	0
			1	PS·PT	PS·PT·PC	PS·PT·PC2		1	PG·PS·PT	PG·PS·PT·PC	PG·PS·PT·PC2		1	PG2·PS·PT	PG2·PS·PT·PC	0
			2	PS·PT2	PS·PT2·PC	0		2	PG·PS·PT2	PG·PS·PT2·PC	0		2	0	0	0
				Cuervos					Cuervos					Cuervos		
				0	1	2			0	1	2			0	1	2
	2	Tejones	0	PS2	PS2·PC	0	Tejones	0	PG·PS2	PG·PS2·PC	0	Tejones	0	0	0	0
			1	PS2·PT	PS2·PT·PC	0		1	PG·PS2·PT	PG·PS2·PT·PC	0		1	0	0	0
			2	0	0	0		2	0	0	0		2	0	0	0

Código del algoritmo:

```
def torneo(n,vc,d):
    """
    int, list(int), int --> double
    OBJ: Devuelve la cantidad que ganaría Javi Potter en caso de que ganen los Grifos a partir de la cantidad apostada d,
    los valores de calidad vc, y el número de partidos necesarios para ganar
    """
    p = []
    for v in vc:
        p.append(v/100)

    tabla = []
    for i in range(n+1):
```

```
    tabla_i = []
    for j in range(n+1):
        tabla_j = []
        for k in range(n+1):
            tabla_k=[]
            for l in range(n+1):
                tabla_k.append(0)
            tabla_j.append(tabla_k)
        tabla_i.append(tabla_j)
    tabla.append(tabla_i)

# La probabilidad de que en un momento dado del torneo (Al comienzo)
# todos los equipos tengan 0 victorias es de 1 ó 100%
tabla[0][0][0][0] = 1

for i in range(n+1):
    for j in range(n+1):
        for k in range(n+1):
            for l in range(n+1):
                if i<n and j<n and k<n and l<n:
                    tabla[i][j][k][l+1] = p[3]*tabla[i][j][k][l]
                    tabla[i][j][k+1][l] = p[2]*tabla[i][j][k][l]
                    tabla[i][j+1][k][l] = p[1]*tabla[i][j][k][l]
                    tabla[i+1][j][k][l] = p[0]*tabla[i][j][k][l]

acumulador = 0
for j in range(n+1):
    for k in range(n+1):
        for l in range(n+1):
            acumulador += tabla[n][j][k][l]

return d/acumulador
```

La complejidad de este algoritmo es de orden $O(n^4)$. Con la mera observación del algoritmo, y por tanto, de los bucles anidados, es trivial deducir que el programa tendrá este orden de complejidad.