Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

Algoritmia y Complejidad

Laboratorio – Sesión 4

Programación dinámica

Laboratorio Jueves 12:00 – 14:00

Grado en Ingeniería Informática – Curso 2018/2019

Eduardo Graván Serrano – 03212337L

Marcos Barranquero Fernández – 51129104N

Ejercicio 4: Dos mochilas

Enunciado

Alí Babá ha conseguido entrar en la cueva de los ciento un mil ladrones, y ha llevado consigo su camello junto con dos grandes alforjas; el problema es que se encuentra con tanto tesoro que no sabe ni qué llevarse. Los tesoros son joyas talladas, obras de arte, cerámica… es decir, son objetos únicos que no pueden partirse ya que entonces su valor se reduciría a cero. Afortunadamente los ladrones tienen todo muy bien organizado y se encuentra con una lista de todos los tesoros que hay en la cueva, donde se refleja el peso de cada pieza y su valor en el mercado de Damasco. Por su parte, Alí sabe la capacidad de peso que tiene cada una de las alforjas. Diseñar un algoritmo que, teniendo como datos los pesos y valor de las piezas, y la capacidad de las dos alforjas, permita obtener el máximo beneficio que podrá sacar Alí Babá de la cueva de las maravillas.

Código

```
def optimizar_valor_mochila(capacidad, array_pesos, array_valores):
    # Obtengo el numero de elementos
    numero_objetos = len(array_pesos)

    # Genero la matriz y la relleno de ceros
    matriz = [[0 for zero in range(capacidad+1)]
              for x in range(numero_objetos+1)]

    # Relleno la matriz con los valores de los arrays
    for fila in range(numero_objetos+1):
        for columna in range(capacidad+1):
            # Si no tienes objetos ni capacidad:
            if fila == 0 or columna == 0:
                matriz[fila][columna] = 0
            elif array_pesos[fila-1] <= columna:
                # Si el nuevo objeto estudiado tiene un peso
inferior a
                # la capacidad máxima de la mochila (columna),
                # compruebo si me sale mejor quedarme como estaba
(es decir,
                # estudiar los objetos que tomaría si no tuviese
ese nuevo objeto)
                # o, por el contrario, añadir ese nuevo objeto y
estudiar qué podría
                # meter en el espacio restante.
                matriz[fila][columna] = max(array_valores[fila-1]
                                            + matriz[fila-
1][columna-array_pesos[fila-1]],
                                              matriz[fila-
1][columna])
            else:
                # Si no puedo meter el nuevo objeto porque su peso
es superior a mi
```

```
                # capacidad en ese momento, me quedo como estaba,
es decir,
                # tomando los objetos que tomé en la iteración
anterior. (Con capacidad-1)
                matriz[fila][columna] = matriz[fila-1][columna]


    # Lista de valores que se han utilizado, usando el peso de los
elementos utilizados.
    lista_usados = []
    # Se guardan en variables locales algunos valores para
    #  poder modificarlos sin alterar el resultado
    resultado = matriz[numero_objetos][capacidad]
    capacidad_temp = capacidad

    # Mirando desde el final de la tabla generada
    for i in range(numero_objetos, 0, -1):
        # Si ya hemos llegado al principio de la tabla, es decir,
si ya hemos examinado
        # todos los posibles valores, salimos del bucle
        if resultado <= 0:
            break
        # Se evalua de donde vienen los valores almacenados en la
tabla
        # Si es igual al valor que se encuentra en la fila
anterior, el bucle continua examinando
        if resultado == matriz[i - 1][capacidad_temp]:
            continue
        # Si no es igual, esto quiere decir que el elemento de la
fila que estamos
        # evaluando ha sido utilizado para obtener el valor óptimo,
        # por lo que se añade a la lista de elementos utilizados.
        else:
            lista_usados.append(array_pesos[i-1])
            # Una vez añadido a la lista, se resta el valor de ese
elemento del resultado
            resultado = resultado - array_valores[i - 1]
            # Y se resta tambien el peso para recolocarnos en la
tabla en la
            # posicion correcta y poder seguir evaluando
            capacidad_temp = capacidad_temp - array_pesos[i - 1]

    return matriz[numero_objetos][capacidad], lista_usados


def mochilero_dos_mochilas(limite_mochila1, limite_mochila2,
array_pesos, array_valores):
    # Fuerzo a que mochila 1 sea la de mayor capacidad.
    temp = limite_mochila1
```

```
    limite_mochila1 = max(lim_mochila1, limite_mochila2)
    limite_mochila2 = max(temp, limite_mochila2)

    # Calculo el valor óptimo que puedo almacenar con la mochila 1:
    optimo_mochila1, lista_usados_uno = optimizar_valor_mochila(
        limite_mochila1, array_pesos, array_valores)

    # Calculo los arrays de los objetos que quedan:
    que_elementos_quedan(lista_usados_uno, array_pesos,
array_valores)

    # Calculo el valor óptimo que puedo almacenar con la mochila 2:
    optimo_mochila2, lista_usados_dos = optimizar_valor_mochila(
        limite_mochila2, array_pesos, array_valores)

    return optimo_mochila1, optimo_mochila2

# La funcion se encarga de eliminar de los arrays de pesos
# y valores aquellos elementos que ya hallan sido utilizados
# a partir de la lista devuelta por la primera llamada a la funcion
de optimización.
def que_elementos_quedan(lista_usados, array_pesos, array_valores):
    n = len(lista_usados)
    for i in range(n):
        indice = array_pesos.index(lista_usados[i])
        array_pesos.pop(indice)
        array_valores.pop(indice)


# Programa principal
array_valores = [1, 6, 18, 22, 28]
array_pesos = [1, 2, 5, 6, 7]
lim_mochila1 = 11
lim_mochila2 = 8


om1, om2 = mochilero_dos_mochilas(
    lim_mochila1, lim_mochila2, array_pesos, array_valores)

print("Optimo con la mochila 1: ", om1)
print("Optimo con la mochila 2 y los elementos restantes: ", om2)
```

Explicación

Tenemos 3 funciones.

La función optimizar_valor_mochila es prácticamente una adaptación del pseudocódigo de las diapositivas de teoría del tema 4, adaptado a listas de Python.

Se crea una matriz de (capacidad de la mochila x número de objetos) celdas, rellena de 0s. La primera fila y primera columna se rellena de 0s, esto es para las filas y columnas con capacidad 0 o 0 objetos tomados. Después, se recorre el resto de la matriz rellenando la celda con el valor máximo entre el valor del objeto evaluado (si cabe) además de la mejor opción con el espacio restante, o el objeto tomado en la máxima capacidad anterior más lo que quepa con la nueva capacidad.

Tras establecer todos los valores de la matriz, queremos devolver una lista con los objetos que hemos tomado en el mejor de los casos, para descartarlos cuando evaluemos la segunda mochila.

Por ello, creamos una lista de objetos usados, y, mirando desde el final de la tabla, vamos tomando los objetos elegidos comprobando su valor y saltando a filas anteriores. Cuando llegamos a cualquier posición con capacidad 0, significa que ya hemos trazado los objetos tomados y por tanto salimos del bucle.

Después, devuelve la casilla con capacidad máxima y evaluando todos los objetos, y la lista de objetos tomados.

Finalmente, la función mochilero_dos_mochilas, que es la función encargada de optimizar ambas mochilas, llama a las anteriores. Primero, fuerzo a que la primera mochila sea la de mayor capacidad.

Después, busco la mejor selección de objetos para la primera mochila. Una vez lo tengo, elimino los objetos usados de las listas de pesos y valores, de forma que no son considerados para la mochila 2.

Tras esto, optimizo la mochila 2 y finalmente devuelvo ambos resultados, es decir, los valores que permiten maximizar el valor para la mochila 1 y la mochila 2.

Ejercicio 7: Bits

Enunciado

Se define una secuencia de bits A como una sucesión A = {a1, a2, …, an} donde cada ai puede tomar el valor 0 o el valor 1, y n es la longitud de la secuencia A. A partir de una secuencia se define una subsecuencia X de A como X = {x1, x2, …, xk}, siendo k ≤ n, de forma que X puede obtenerse eliminando algún elemento de A pero respetando el orden en que aparecen los bits; por ejemplo, si A = {1, 0, 1, 1, 0, 0, 1} podríamos obtener como subsecuencias {1, 1, 1, 0, 1}, {1, 0, 1} o {1, 1, 0, 0} entre otras, pero nunca se podría conseguir la subsecuencia {1, 0, 0, 1, 1}. Dadas dos secuencias A y B, se denomina a X una subsecuencia común de A y B cuando X es subsecuencia de A y además es subsecuencia de B. (aunque puede que se hayan obtenido quitando distintos elementos en A que B, e incluso distinta cantidad). Suponiendo las secuencias A = {0, 1, 1, 0, 1, 0, 1, 0} y B = {1, 0, 1, 0, 0, 1, 0, 0, 1}, una subsecuencia común sería X = {1, 1, 0, 1}, pero no podría serlo X = {0, 1, 1, 1, 0}. Se desea determinar la subsecuencia común de dos secuencias A y B que tenga la longitud máxima, para lo que se pide • explicar con detalle la forma de resolver el problema, y • hacer un algoritmo de Programación Dinámica que obtenga la longitud máxima posible y una secuencia común de dicha longitud.

Código

```
import numpy as np


def es_subcadena(cadena, subcadena):
    """ Dada una subcadena y una cadena, devuelve True si la
subcadena es es en efecto subcadena de cadena. Si no, False. """
    """
    Funciona comparando los bits de la subcadena con los de la
cadena.
    Para cada bit, busca en la cadena la posición más cercana a 0
con ese bit.
    Cuando lo encuentre, guarda esa posición nueva y busca el
siguiente bit desde la posición nueva
    hasta el final, más cercano al comienzo. Y así con todos los
bits. Si hace un index out of range,
    es que no se ha encontrado un bit de la subcadena en lo que
quedaba de la cadena, por tanto, no es subcadena. """

    posicion_en_cadena = 0
    for bit in subcadena:
        try:
            while(bit != cadena[posicion_en_cadena]):
                posicion_en_cadena += 1
            posicion_en_cadena += 1
        except:
            return False
    return True


def imprime_matriz(matriz):
    """ Tira de numpy pa dibujar la matriz en la terminal. """
    print(np.matrix(matriz))


def crear_matriz(cadena_a, cadena_b):
    """ Dadas dos cadenas, te inicializa la matriz
    de programación dinámica con todo cerapios y la
    primera fila y columna completada. Se podría hacer
    directamente en la otra función, pero te puedes volver loco
    con tanto if/elif/else, así que hemos decidido separarlo. """

    # Inicio matriz a ceros:
    matriz = [[0 for zero in range(len(cadena_b))]
              for x in range(len(cadena_a))]

    # Para cada posición matriz[x][0] o matriz[0][x],
    # asigno su valor de subcadena máximo, que viene siendo
```

```
    # que el bit 0 de una cadena esté contenido en la otra al menos
1 vez.
    for fila in range(len(cadena_a)):
        for columna in range(len(cadena_b)):
            if (fila == 0):
                if(cadena_a[0] in cadena_b[0:(columna+1)]):
                    matriz[fila][columna] = 1
                else:
                    matriz[fila][columna] = 0
            elif (columna == 0):
                if(cadena_b[0] in cadena_a[0:(fila+1)]):
                    matriz[fila][columna] = 1
                else:
                    matriz[fila][columna] = 0
    return matriz


def get_max_len(cadena_a, cadena_b):
    """Esta es la función principal."""

    # creo la matriz de 0s con fila 0 y columna 0 inicializada.
    matriz = crear_matriz(cadena_a, cadena_b)

    # para cada posición
    for fila in range(len(cadena_a)):
        for columna in range(len(cadena_b)):
            # descarto las filas ya hechas
            if(fila == 0 or columna == 0):
                continue
            # La magia es esta:
            # Si en anteriores iteraciones no he podido hacer una
subcadena de la longitud de
            # la cadena que estudio, es porque me ha faltado el
último bit.
            # Por tanto, si el último bit resulta ser igual que el
bit con el que cuento en la
            # siguiente iteración, podré añadirlo y por tanto
aumentar en 1 la longitud de la
            # subcadena máxima.
            elif(matriz[fila][columna-1] != (fila+1) and
cadena_b[columna] == cadena_a[fila]):
                matriz[fila][columna] = matriz[fila][columna-1] + 1

            # Si no, el tamaño máximo será el tamaño máximo
anterior.
            elif(matriz[fila][columna] < matriz[fila][columna-1]):
                matriz[fila][columna] = matriz[fila][columna-1]
    # Imprimo la matriz pa que se vea en terminal.
    imprime_matriz(matriz)
```

```
    # Y devuelvo la longitud máxima para las longitudes de las
cadenas dadas.
    return matriz[len(cadena_a)-1][len(cadena_b)-1]


def binario(numero):
    """ Dado un nº devuelve su representación en binario"""
    return str(bin(numero))[2:]


def es_doble_subcadena(subcadena, a, b):
    """ Dadas dos cadenas a y b, una cadena es subcadena de ambas
    si es subcadena de a y subcadena de b. """
    return es_subcadena(a, subcadena) and es_subcadena(b,
subcadena)


def get_ejemplo(cadena_a, cadena_b, max_len):
    """ una vez calculada la longitud máxima de cadena para
    unas cadenas dadas, genero todas las posibles cadenas con
    esa longitud, y las pruebo una por una para ver cuales
    son subcadenas de las cadenas a y b. """

    posibles_combinaciones = []
    for i in range(2**max_len):
        posibles_combinaciones.append(binario(i))
    posibles2 = []

    for posible in posibles_combinaciones:
        if(len(posible) != max_len):
            posible = str("0"*(max_len-len(posible))) +
str(posible)
        posibles2.append(posible)
    resultados = []
    for posible in posibles2:
        if(es_doble_subcadena(posible, A, B)):
            resultados.append(posible)
    return resultados

# Programa principal


A = "01101010"
B = "101001001"


print(es_doble_subcadena("010101", A, B))
```

```
resultados = get_ejemplo(A, B, get_max_len(A, B))

print(resultados)
```

Explicación

Primero, hemos creado dos funciones de verificación. La función “es_subcadena” recibe una cadena y subcadena de bits, y devuelve True si la subcadena es -efectivamente- subcadena de bits de la cadena proporcionada. La función “es_doble_subcadena” recibe dos posibles subcadenas de una cadena de bits, y apoyándose en la función anterior descrita, devuelve True si las subcadenas son efectivamente subcadenas posibles de la cadena proporcionada.

Segundo, hemos creado una función llamada “get_max_len” que, mediante programación dinámica, devuelve la máxima longitud posible de una cadena.

Se crea una matriz rellena de 0s, de tamaño (longitud de subcadena1, longitud de subcadena2). Para las casillas de fila 0 o columna 0, se rellena con 0s.

Para el resto, el tamaño máximo de la cadena será el máximo entre el valor de la posición con fila o columna anterior (es decir, el de una subcadena sin considerar ese bit), o, en caso de que el nuevo bit considerado se replique también en la cadena y la longitud de esta cadena no sea la máxima posible, será el valor anterior + 1.

Es decir, si en la iteración anterior no he podido hacer la subcadena tan larga como la cantidad de bits que estoy estudiando en esa cadena, es porque el último bit no era igual. Si ahora son iguales, puedo añadirlo.

Finalmente devolvemos el valor de la última casilla, que será el tamaño máximo de la cadena de bits.

Para obtener un ejemplo de cadena, se crea la función get_ejemplo, que recibe dos cadenas y la máxima longitud de subcadena posible, que se calcula con la función anterior descrita. Se crean cadenas de tamaño (máxima longitud de subcadena posible) bits, y se prueban cuáles son subcadenas con la función es doble subcadena. Si efectivamente son subcadenas, se añaden a una lista de subcadenas posibles encontradas, que luego se devuelve.