s

Instituto Politécnico Nacional
Escuela Superior de Computo

Análisis de algoritmos

Sandra Díaz Santiago

Practica 8: Programación dinámica

García González Aarón Antonio & Vallejo Serrano Ehecatzin

Noviembre 29, 2019

Índice

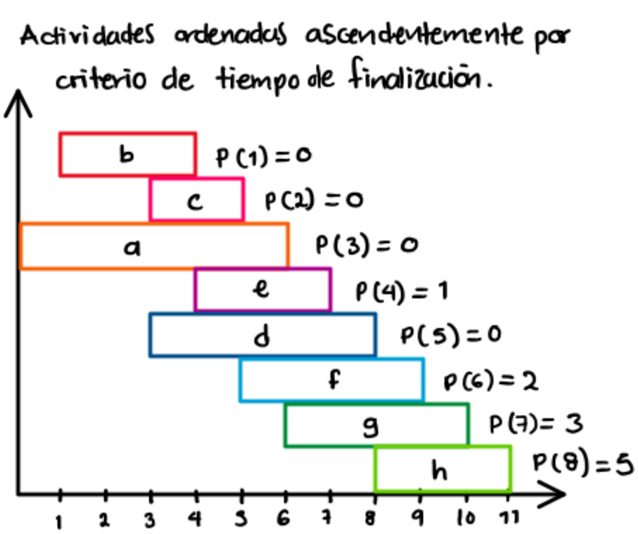
Ejercicio 1

Considera el problema de la planificación de actividades, pero ahora cada actividad tiene un valor vi, además de los tiempos de inicio si y los tiempos de finalización fi. El objetivo ahora ya no es maximizar el número de actividades en un intervalo de tiempo, sino maximizar el valor total de las actividades seleccionadas. Es decir, se desea elegir un conjunto de actividades compatibles A, tales que $\sum a_{k \epsilon A} v_k$ sea el máximo

[a]. Escribe el pseudocódigo del algoritmo que utiliza programación dinámica para resolver el problema.
[b]. Usa un ejemplo propuesto por ti, para mostrar el funcionamiento de tu algoritmo.
[c]. Calcula la complejidad de tu algoritmo, indicando cuál es el tamaño del problema.
[d]. Implementa tu algoritmo y realiza las pruebas necesarias, para asegurarte que funciona correctamente.

Definiremos el siguiente ejemplo:

ID actividad	Inicio	Final	Valor
b	1	4	2
c	3	5	4
a	0	6	3
e	4	7	3
d	3	8	1
f	5	9	4
g	6	10	1
h	8	11	3



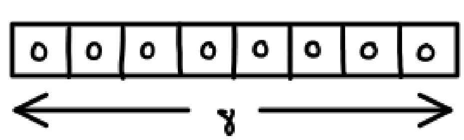
Con base al ejemplo anterior vamos a verificar el funcionamiento de nuestro programa:

Los p(j), indican la actividad inmediata anterior que finaliza cuando la actividad j comienza.

1. Lo primero que realiza el algoritmo es ordenar las actividades ascendentemente con base al tiempo de finalización de cada actividad, como se muestra en la figura.

Para ir almacenando los valores de acumulados de máximo valor, definimos un arreglo e inicializamos en cero cada una de sus posiciones, este arreglo será de tamaño j actividades.

En el diagrama de arriba podemos representar las Pj's como cero cuando no encuentra alguna actividad sin traslape que sea predecesora, en el programa en vez de colocar 0's ponemos -1 en su lugar.

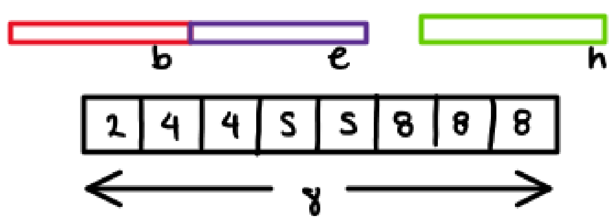


Con la función **binarySearch(actividades, i)** buscamos el correspondiente pj por cada actividad, luego si este pj es distinto de -1, es decir tiene una actividad adyacente a su inicio, entonces consideraremos su valor y le sumaremos el de la actividad pj que encontramos, entonces nuestro vector solución en la posición j, será igual al máximo:

a) El valor de la actividad en cuestión mas el valor de su actividad pj
b) El vector solución en una posición anterior

Comenzamos con el ejemplo:

- La primera actividad es b, donde su pj es -1, en el vector solución será igual a su mismo valor, es decir 2
- La segunda actividad es c, donde si pj es -1, en el vector solución será igual a su mismo valor, es decir 4
- La tercera actividad es a, donde su pj es -1, en el vector solución será igual a el máximo entre su mismo pj y el vector solución en una posición menor, es decir máximo (3,4), entonces vector solución de dicha posición será igual con 4
- La cuarta posición es e, donde su pj es 1, en el vector solución será igual al máximo (3+2,4), es decir será igual con 5
- La quinta posición es d, donde su pj es -1, en el vector solución será igual al máximo entre (1,5), es decir será igual con 5
- La sexta posición es f, donde su pj es 2, en el vector solución será igual al máximo (4+4,5), es decir será igual con 8
- La séptima actividad, donde su pj es 3, en el vector solución será igual al máximo (1+3, 8), es decir será igual con 8
- La octava actividad, donde su pj es 5, en el vector solución será igual al máximo (3+1, 8), es decir será igual con 8



Calculo de la complejidad del algoritmo:

La implementación del algoritmo considero que es la optima, ya que dentro de un ciclo que itera n-1 veces y dentro de este ciclo se encuentra una búsqueda binaria, cuyo costo es O(log n), entonces el costo de dicho algoritmo es a lo mas O([n-1]Log n) -> nLogn – Logn, esto se logro gracias a la técnica de memoización, que vamos tomando valores que previamente ya habíamos calculado y nos evitamos llamadas recursivas que ralentizan el programa.

```
class Actividad:
    def __init__(self, inicio, finalizacion, valor, identificador):
        self.inicio  = inicio
        self.finalizacion = finalizacion
        self.valor  = valor
```

```
        self.id = identificador

    def mostraraActividad(self):
        print("ID de actividad: ", self.id)
        print("Hora de inicio: ", self.inicio)
        print("Hora de finalización: ", self.finalizacion)
        print("Valor de actividad: ", self.valor, "\n")



# Una función basada en la búsqueda binaria para encontrar la ultima actividad
# (antes de la actividad actual) que no entre en conflicto con la actividad actual.
# "index" es el índice de la actividad actual. Esta función devuelve -1 si todos
# las actividades anteriores al índice entran en conflicto con ella. Las actividades
# de matriz [] se ordenan en orden creciente de tiempo de finalización.
def binarySearch(actividades, inicio_index):

    # Inicializamo 'lo'y 'hi' para busqueda binaria
    lo = 0
    hi = inicio_index - 1

    # Realizar búsqueda binaria iterativamente
    while lo <= hi:
        mid = (lo + hi) // 2
        if actividades[mid].finalizacion <= actividades[inicio_index].inicio:
            if actividades[mid + 1].finalizacion <= actividades[inicio_index].inicio:
                lo = mid + 1
            else:
                return mid
        else:
            hi = mid - 1
    return -1

# La función principal que devuelve el máximo posible.
#   valor de una variedad dada de actividades
def Planificar(actividades):

    # Ordenar actividades según el tiempo de finalización
    actividades = sorted(actividades, key = lambda j: j.finalizacion)
```

```
    # Creamos una matriz para almacenar soluciones de subproblemas. table[i]
    # almacena el valor para trabajos hasta arr[i] (incluendo arr[i])
    n = len(actividades)
    table = [0 for _ in range(n)]
    tareas = []
    aux = []

    table[0] = actividades[0].valor; # la primera por default no tiene predecesor

    # Rellenar entradas en la tabla [] usando la propiedad recursiva
    for i in range(1, n):
        # Encuentra valor incluyendo la actividad actual
        valor_inicial = actividades[i].valor
        l = binarySearch(actividades, i) # busca el predecesor anterior mas cercano

        if (l != -1): # si tiene predecesor
            valor_inicial += table[l];

        # Almacenar máximo de incluir y excluir
        table[i] = max(valor_inicial, table[i - 1])



    return table


def generarListaActividades():
    resultado = []
    actividades_sin_formato = leerDatos("input_1.txt")
    for act in actividades_sin_formato:
        resultado.append(Actividad(int(act[0]),int(act[1]),int(act[2]),str(act[3])))
    return resultado


# Código de controlador para probar la función anterior
lista_de_actividades_2 = generarListaActividades()
```

```python
print("Valor optimo: ")
print(Planificar(lista_de_actividades_2))
```

Ejecución

```
MacBook-Pro-de-Aaron:p8 aarongarcia$ python3 index_1.py
Valor optimo:
[2, 4, 4, 5, 5, 8, 8, 8]
```

Datos tomados a partir de un archivo:

```
index_1.py    input_1.txt
p8 > input_1.txt
1 4 2 b
3 5 4 c
0 6 3 a
4 7 3 e
3 8 1 d
5 9 4 f
6 10 1 g
8 11 3 h
```

Donde leemos, (inicio, fin, valor, identificador)

Ejercicio 2

Diseña un algoritmo que utilice programación dinámica, para establecer cuál es la forma óptima de multiplicar n matrices. Tu algoritmo debe recibir como entrada el número de matrices a multiplicar n y las dimensiones de cada una m0, m1, . . . , mn, en la secuencia adecuada, es decir m0×m1 serán las dimensiones de la matriz A1, m1×m2 serán las dimensiones de la matriz A2 y así sucesivamente. Tu algoritmo debe mostrar la tabla de memoización y la secuencia de multiplicación seleccionada, así como el costo de dicha secuencia.

Pseudocodigo

```
Mult_cadena_matrices(dimenciones)
    N ← longitud de dimenciones – 1
    Desde i = 1 hasta N
        M[i][i] = 0
    Finde ciclo
    Desde l =2 hasta n
        Desde i = 1 hasta n – l +1
            J ← i + l -1
            M[i][j] ← infinito
            Desde k = i hasta j -1
                Q ← m[i][k] + m[k+1][j] + p_{i-1} p_k p_j
                Si Q < m[i][j] entonces
                    M[i][j] ← q
                    S[i][j] ← k
```

Tenemos las siguientes matrices:

Matriz	A1	A2	A3	A4
Dimensión	30x35	35x15	15x5	5x10

Entonces el vector de dimenciones, es:

p = [30, 35, 15, 5, 10]

$A_1A_2A_3 \quad A_4$

30 x 15

15 x 5

30 x 5 x 10

	1	2	3	4
1	0	15,750	7,875	9,375
2		0	2,625	4,375
3			0	750
4				0

Si multiplicamos la matriz 1 por la 2, tenemos 2 puntos de corte, es decir (1,2) = 30 x 35 x 15 = 15,750

Si ahora multiplicamos la 2 por la 3, entonces:

(2,3) = 35 x 15 x 5 = 2,625

(3,4) = 15 x 5 x 10 = 750

Ahora, multiplicar de la 1 a la 3, hay 2 puntos de corte, por lo que hay 2 alternativas:

- A1 (A2·A3), donde $A_2 \cdot A_3$ ya la conocemos:
  = 0 + 2,625 + 35 x 5 x 30 = 2,625 + 5,250 = 7,875

ó

- (A1·A2) A3, donde A1·A2 ya la conocemos:
  = 15,750 + 0 + 30 x 15 x 5 = 15,750 + 2,250 = 18,000

De estos dos anteriores p(1,3) = min (7,875, 18,000) = 7,875

Ahora multiplicar de 2 a 4, de nuevo hay 2 puntos de corte:

- $A_2 A_3 \cdot A_4$, $A_2 \cdot A_3$ = 2625
  = 2,625 + 0 + 35 x 5 x 10 = 2,625 + 1,750 = 4,375

ó

- $A_2 \cdot A_3 A_4$, $A_3 \cdot A_4$ = 750
  = 0 + 750 + 15 x 10 x 35 = 750 + 5,250 = 6,000

De estos anteriores, p(2,4) = min(4,375, 6,000) = 4,375

Finalmente multiplicar de 1 a 4, es decir hay 3 puntos de corte:

- $A_1 \cdot A_2A_3A_4$, A_2A_4 = 4,375
  = 0 + 4,375 + 35 x 10 x 30 = 4,375 + 10,500 = 14,875

- $A_1A_2 \cdot A_3A_4$, $A_1 \cdot A_2$ = 15,750 , A_3A_4 = 750
  = 15,750 + 750 + 30 x 15 x 10 = 15,750 + 750 + 4,500 = 21,000

- $A_1A_2A_3 \cdot A_4$, $A_1 \cdot A_3$ = 7,875
  = 7,875 + 0 + 30 x 5 x 10 = 7,875 + 1,500 = 9,375

De los anteriores p(1,4) = min (14,875, 21,000, 9,375)
= 9,375

15) Fin de condicional
16) Fin de ciclo
17) Fin de ciclo
18) Fin de ciclo
19) return M,S

En la linea 6, se trata del tamaño del intervalo, es decir el número de matrices
En la linea 7, primer elemento del intervalo, (1,2),(2,3),(4,5),(1,3),(2,4),(3,5)…
En la linea 8, Ultimo elemento del intervalo
En la linea 10, puntos de corte
En la linea 14, esta linea la usaremos para crear la misma matriz, pero con las posiciones en donde se realizaron los cortes, asi poder decir cual es el orden que se tiene que realizar para multiplicar la cadena de matrices de manera optima.
Y por cada uno de los anteriores escogemos el minimo

Entonces, la cantidad de subproblemas es $\frac{n(n-1)}{2}$ (que se refiere a la triangular que nos queda), es decir que la cantidad de subproblemas es cuadratico O(n^2), y cada problema nos cuesta resolver O(n) porque cada subproblema es escoger entre los posibles puntos de corte donde en el peor de los casos tenemos n-1 puntos de corte, y la eficiencia total del programa es O(n^3), esto es porque juntamos las dos complejidades anteriores debido a que se encuentran en bubles anidados (3 ciclo for), pues por ello es que llegamos a dicha complejidad.

```
def producto_matricial(p):
    """Return m and s.

    m[i][j] e sel numero minimo de multiplicaciones escalares que se necesitan realizar
        con el producto de matrices A(i), A(i + 1), ..., A(j).

    s[i][j] es el índice de la matriz después del cual el producto se divide en un
     paréntesis óptimo del producto matriz.

    p[0... n] es una lista tal que la matriz A (i) tiene dimensiones p [i - 1] x p [i].
    """
    length = len(p) # len(p) = numero de matrices + 1

    # m[i][j] es el número mínimo de multiplicaciones necesarias para calcular el
    # producto de matrices A(i), A(i+1), ..., A(j)
    # s[i][j] es la matriz después de la cual el producto se divide en el mínimo
    # numero de multiplicaciones necesarias
    m = [[-1]*length for _ in range(length)]
    s = [[-1]*length for _ in range(length)]
```

```
    vaproductos_matriz(p, 1, length - 1, m, s)

    return m, s



def vaproductos_matriz(p, start, end, m, s):
    """Return minimum number of scalar multiplications needed to compute the
    product of matrices A(start), A(start + 1), ..., A(end).

    El número mínimo de multiplicaciones escalares necesarias para calcular el
    producto de matrices A(i), A(i + 1), ..., A(j) esta almacenado en m[i][j].

    El índice de la matriz después del cual el producto anterior se divide en un óptimo
    el paréntesis se almacena en s[i][j].

    p[0... n] es una lista tal que la matriz A (i) tiene dimensionesp[i - 1] x p[i].
    """
    if m[start][end] >= 0:
        return m[start][end]

    if start == end:
        q = 0
    else:
        q = float('inf')
        for k in range(start, end):
            temp = vaproductos_matriz(p, start, k, m, s) \
                   + vaproductos_matriz(p, k + 1, end, m, s) \
                   + p[start - 1]*p[k]*p[end]
            if q > temp:
                q = temp
                s[start][end] = k

    m[start][end] = q
    return q



def imprimir_palentizacion(s, start, end):
    """ Imprimimos el paréntesis óptimo del producto matriz A (inicio) x
```

```
    A(inicio + 1) x ... x A(final).

    s[i][j] es el índice de la matriz después del cual el producto se divide en un
    paréntesis óptimo del producto matriz.
    """
    if start == end:
        print('A[{}]'.format(start), end='')
        return

    k = s[start][end]

    print('(', end='')
    imprimir_palentizacion(s, start, k)
    imprimir_palentizacion(s, k + 1, end)
    print(')', end='')


n = int(input('Teclea el número de matrices: '))
p = []
for i in range(n):
    temp = int(input('Teclea el numero de filas de la matriz {}: '.format(i + 1)))
    p.append(temp)
temp = int(input('Teclea el número de columnas de la matrix {}: '.format(n)))
p.append(temp)

m, s = producto_matricial(p)
print('#De multiplicaciones escalares necesarias:', m[1][n])
print('Palentización óptima: ', end='')
imprimir_palentizacion(s, 1, n)
print("\n")
```

Ejecución

```
MacBook-Pro-de-Aaron:p8 aarongarcia$ python3 index_2.py
Teclea el número de matrices: 4
Teclea el numero de filas de la matriz 1: 30
Teclea el numero de filas de la matriz 2: 35
Teclea el numero de filas de la matriz 3: 15
Teclea el numero de filas de la matriz 4: 5
Teclea el número de columnas de la matrix 4: 10
-------------------------- Matriz ---------------------------
[-1, -1, -1, -1, -1]
[-1, 0, 15750, 7875, 9375]
[-1, -1, 0, 2625, 4375]
[-1, -1, -1, 0, 750]
[-1, -1, -1, -1, 0]
-------------------------------------------------------------
#De multiplicaciones escalares necesarias: 9375
Palentización óptima: ((A[1](A[2]A[3]))A[4])
```