Análisis de Algoritmos

Tarea 3 - Exponenciación y Cadenas de Matrices

Gabriela N. Góngora Svartzman, Karen L. Poblete Rodríguez,
Salvador B. Medina Maza, Victor R. Martinez Palacios

20/09/2013

PARTE 1. Exponenciación

¿Cuál es la forma óptima de calcular $x^{77,}$ x^{195} y x^{631}, con dos posiciones de memoria? Encontrar un método.

Hint: Uso en Public Key Encryption. El cómputo requerido para descifrar claves públicas, es el cálculo de $x^e mod N$ *, para* x *,* e *y* N *enteros positivos de hasta 1000 dígitos.*

La solución óptima con sólo dos posiciones de memoria es meramente teórica, ya que en la práctica no se podría realizar dicho exponenciación con sólo dos espacios de memoria. Puede que el programa utilice sólo dos variables, pero estas variables se traducen a nivel ensamblador en muchos más espacios de memoria (si consideramos registros, ciclos de reloj e inclusive los re-arreglos de código que podría hacer el compilador).

La solución teórica es la siguiente. M1 = primer espacio de memoria. M2 = segundo espacio de memoria. n = número total de multiplicaciones que llevó realizar dicha exponenciación.

A. x^{77}. n=8.

M1	M2
x	
	x * x = x^2
	$x^2 * x^2 = x^4$
	$x^4 * x^4 = x^8$
$x^8 * x = x^9$	
	$x^8 * x^9 = x^{17}$
	$x^{17} * x^{17} = x^{34}$
$x^{34} * x^9 = x^{43}$	
	$x^{43} * x^{34} = x^{77}$

B. x^{195}. n = 9

M1	M2
x	
	x * x = x^2
$x^2 * x = x^3$	
	$x^3 * x^3 = x^6$
	$x^6 * x^6 = x^{12}$
	$x^{12} * x^{12} = x^{24}$
	$x^{24} * x^{24} = x^{48}$
	$x^{48} * x^{48} = x^{96}$
$x^{96} * x^3 = x^{99}$	
	$x^{99} * x^{96} = x^{195}$

C. x^{631}. n = 12

M1	M2
x	
	x * x = x^2
	$x^2 * x = x^3$
	$x^3 * x^3 = x^6$
$x^6 * x = x^7$	
	$x^7 * x^6 = x^{13}$
	$x^{13} * x^{13} = x^{26}$
	$x^{26} * x^{26} = x^{52}$
	$x^{52} * x^{52} = x^{104}$
	$x^{104} * x^{104} = x^{208}$
$x^{208} * x^7 = x^{215}$	
$x^{215} * x^{208} = x^{423}$	
	$x^{423} * x^{208} = x^{631}$

Para cualquier número de espacios de memoria podemos resolver el problema de manera práctica. En el área de criptografía se requiere mucho de métodos de exponenciación para

calcular e inclusive proteger sus llaves (públicas y privadas). Existen técnicas básicas *(repeated square and multiply algorithms)*, de exponente fijo y de base fija. Investigamos las técnicas básicas, las cuales permiten mayor flexibilidad e implementamos tres algoritmos que nos ayudaron a calcular los valores para $x^{77,}$ x^{195} y x^{631}. Utilizamos X = 2. para realizar las pruebas. Todos los algoritmos se programaron en Python.

Algoritmo 1. RLbinary

La función RLbinary(g,e), donde g es la base y e es el exponente, respeta la restricción de dos espacios de memoria. Trata de optimizar las multiplicaciones haciendo modulos 2 del valor del exponente. A continuación se muestra el código de la función con una explicación sencilla de lo que se realiza paso a paso.Se cuentan con dos operaciones básicas, multiplicar A*S o multiplicar S*S.

```
def RLbinary(g,e):
   A=1                          %Inicializamos el primer espacio de memoria A en 1
   S=g                          %Ponemos en S el valor de la base g.

   while (e!=0):                %WHILE e (exponente) sea diferente de 0
      if ((e%2)!=0):            %IF Módulo de e diferente de 0
         A=A*S                  % A lo que había en A se multiplica por lo que hay en S.

      e=int(e/2)                %e es función piso de e/2.
      if e!=0:                  %IF e diferente de 0
         S=S*S                  %S vale su cuadrado.
   return (A)                   %END WHILE y RETURN
```

Algoritmo 2. LRbinary

La función LRbinary(g,e), donde g es la base y el exponente, utiliza el valor del exponente en binario para saber cuándo elevar al cuadrado loque trae en el acomulador o agregar el valor de la base. En este sistema el valor de la base no se modifica, es decir, en un espacio de memoria se guarda el valor original de g y sólo se utiliza para aumentar el valor de A.

```
def LRbinary(g,e):
   A=1                          %Inicialización de variable A. (Representa un espacio de memoria).
   E=parseBin(e)                % Valor de exponente en binario.

   for i in range(len(E)):      %FOR Para cada valor en E
      A=A*A                     %A es igual a A al cuadrado.

      if (E[i]=='1'):           %IF Sí E(i) == 1 en su representación binaria.
         A=A*g                  %A es igual a A*g, g siendo la base
```

```
    return (A)              %END IF, END FOR.
```

Algoritmo 3. modularExp

La función modularExp(a,b,n), donde a es la base, b el exponente que se debe calcular y n es un número mayor a b y primo utilizado para hacer el módulo. Es decir este método realiza operacionesmóduloo sobre el valor del exponente además de utilizar el valor del exponente en código binario.

```
def modularExp(a,b,n):
    c=0                     %Inicialización de variable c (Representa un espacio de memoria).
    d=1                     %Inicialización de variable d. (Representa un espacio de memoria).
    bk=parseBin(b)                  %Valor de exponente en binario.
    temp=d                  %Variable temporal para ir guardando el valor de a^b

    for i in range(len(bk)):
        c=2*c               %c por dos.
        d=int((d*d)%n)      % Se obtiene el módulo n del cuadrado de d.
        temp=temp*temp      %Realiza misma operación que arriba pero sin el módulo para obtener su
valor

        if(bk[i]=='1'):     %IF Sí bk(i) == 1 en su representación binaria.
            c=c+1           % c se incrementa en 1.
            d=int((d*a)%(n))    % Se obtiene el módulo n de la multiplicación de “a” con “d”.
            temp=temp*a     %Realiza misma operación que arriba pero sin el módulo para obtener su
valor
      return (temp)         %END IF, END FOR.
```

Resultados

Se compararon los resultados de los tres algoritmos: RLbinary, LRbinary y modularExp. A continuación se presenta la tabla para calcular los siguientes exponentes: $x^{77,}$ x^{195} y x^{631}

Resultados para x^{77}

```
 LR BINARY
TIME:  24 microseconds
Result: 151115727451828646838272
Number of mult = 11
Add Chain =  [0, 1, 2, 4, 8, 9, 18, 19, 38, 76, 77]

 RL BINARY
RLbinary  151115727451828646838272
TIME:  16 microseconds
```

```
mults  10
Add Chain =  [1, 2, 4, 5, 8, 13, 16, 32, 64, 77]

 Modular Exponentiation
TIME:  31 microseconds
Result: 151115727451828646838272
Number of mult = 11
Add Chain =  [0, 1, 2, 4, 8, 9, 18, 19, 38, 76, 77]
```

Resultados para x^{195}

```
RL BINARY
TIME:  16 microseconds
Result: 5021681388309344611068631538566133132881884355571227610316 8
Number of mult = 11
Add Chain =  [1, 2, 3, 4, 8, 16, 32, 64, 67, 128, 195]

 Modular Exponentiation
TIME:  31 microseconds
Result: 5021681388309344611068631538566133132881884355571227610316 8
Number of mult = 12
Add Chain =  [0, 1, 2, 3, 6, 12, 24, 48, 96, 97, 194, 195]

 LR BINARY
TIME:  26 microseconds
Result: 5021681388309344611068631538566133132881884355571227610316 8
Number of mult = 12
Add Chain =  [0, 1, 2, 3, 6, 12, 24, 48, 96, 97, 194, 195]
```

Resultados para x^{631}

```
 Modular Exponentiation
TIME:  41 microseconds
Result:
891101683129335003640853829238338149393208692821984361441248538652202181095444802051936095960424101519266076088592657677868887640893640234033722914008244958642967709835989248063061365673164 8
Number of mult = 17
Add Chain =  [0, 1, 2, 4, 8, 9, 18, 19, 38, 39, 78, 156, 157, 314, 315, 630, 631]

 LR BINARY
TIME:  32 microseconds
Result:
891101683129335003640853829238338149393208692821984361441248538652202181095444802051936095960424101519266076088592657677868887640893640234033722914008244958642967709835989248063061365673164 8
Number of mult = 17
Add Chain =  [0, 1, 2, 4, 8, 9, 18, 19, 38, 39, 78, 156, 157, 314, 315, 630, 631]

 RL BINARY
```

```
TIME:  20 microseconds
Result:
891101683129335003640853829238338149393208692821984361441248538652202181095444802051936095960424101519266076088592657677868887640893640234033722914008244958642967709835989248063061365673164 8
Number of mult = 16
Add Chain =  [1, 2, 3, 4, 7, 8, 16, 23, 32, 55, 64, 119, 128, 256, 512, 631]
```

La siguiente tabla recopila la información de los resultados obtenidos para los tres valores calculados en tiempo y número de multiplicaciones.

Exponente	x^{77}	x^{195}	x^{631}
LRbinary			
Tiempo (microsegundos)	24	26	32
Num. Mult	11	12	17
RLbinary			
Tiempo (microsegundos)	16	16	20
Num. Mult	10	11	16
Modular Exp			
Tiempo (microsegundos)	31	31	41
Num. Mult	11	12	17

El algoritmo con mejor desempeño de los tres presentados fue RLbinary, ya que toma menos tiempo de ejecución y menos multiplicaciones, que los otros dos, para llegar al resultado correcto. Hay que tomar en cuenta que en el caso de los otros dos algoritmos, lo que más les está costando son las divisiones u operaciones módulo, las cuales llevan más ciclos de reloj en realizarse, es decir, son más costosas a nivel cómputo. Existe un problema entre el tiempo de ejecución del programa para calcular el número de exponenciaciones y la cantidad de multiplicaciones que se requieren. No podemos tener ambos, siempre vamos a terminar sacrificando un aspecto. Existen otros algoritmos que hacen lo mismo, pero son más complicados o son más costosos computacionalmente, aunque realicen menos multiplicaciones. Se vuelve una cuestión de que queremos sacrificar y de que queremos lograr con el algoritmo que utilizamos.

Que el algoritmo RLbinary sea mejor que los otros no quiere decir que sea el óptimo, este es un problema es NP-hard, por eso se hacen buenas aproximaciones que ahorran tiempo

y espacio de memoria. En tiempos el algoritmo más rápido también fue RLbinary como la tabla lo muestra.

PARTE 2. Encadenación de Matrices

Producto de cadena de matrices. Dada las matrices $A_1 \times A_2 \times ...A_n$, donde las matrices tienen dimensiones compatibles - pero no son cuadradas - ¿en qué orden deben ser multiplicadas para minimizar la cantidad de multiplicaciones requeridas?

$A_1 = 100 \times 4$
$A_2 = 4 \times 50$
$A_3 = 50 \times 20$
$A_4 = 20 \times 100$
Exponer ejemplos, mejor y peor caso, y describir el método empleado.

Para resolver este problema recurrimos a la programación dinámica, por lo cual necesitamos nombrar sus objetivos:

1. Caracterizar la estructura de una solución óptima
2. Definir recursivamente el valor de una solución óptima
3. Computar el valor de la solución óptima
4. Construir una solución óptima de la información computada

En el resto del documento seguiremo la siguiente convención:

$A_{i...j}$ es la matriz resultante del producto de las matrices $A_iA_{i+1}...A_j$

Caracterizando el problema podemos asumir que $A_{i...j}$ se puede dividir en $A_{i...k}$ y $A_{k+1...j,}$, donde lo que se busca optimizar es el costo computacional que en nuestro caso es el número de multiplicaciones que existen en las subcadenas de matrices. Por lo tanto el costo lo podemos definir de la siguiente manera:

$$\mathbf{C = C(A_{i...k}) + C(A_{k+1...j}) + C(A_{i...k} \times A_{k+1...j})}$$

donde:

$C(A_{i...k})$:	Costo de computar $A_{i...k}$
$C(A_{k+1...j})$:	Costo de computar $A_{k+1...j}$
$C(A_{i...k} \times A_{k+1...j})$:	Costo del producto entre $A_{i...k}$ y $A_{k+1...j}$

Por lo que el problema original se puede dividir en dos sub-problemas:

Optimizar $C(A_{i...k})$ y Optimizar $C(A_{k+1...j})$

Solución recursiva

De esta forma para poder resolver los subproblemas previamente establecido, se tiene que determinar la posición k que indica el punto de quiebre óptimo de la cadena de matrices para obtener el costo mínimo de $A_i...A_j$.

Dentro de la solución propuesta, se va a tener una aproximación *bottom-up* tabular, es decir, se explorarán las posibles subcadenas para calcular el número de multiplicaciones que éstas requieren y se irá guardando el valor mínimo de multiplicaciones en una matriz **m** donde:

m[i,j] almacenará el número mínimo de multiplicaciones escalares

siendo *i* el índice del inicio de la subcadena y *j* el fin de la misma. Por lo tanto, el algoritmo almacenará en m[1,n] el valor mínimo de toda la cadena, ya que esta posición dentro de la matriz representa el valor óptimo de la cadena que va de 1 hasta n, donde n es la longitud de la cadena de matrices.

De la misma manera también se establece una matriz **s** que irá almacenando el valor del punto de quiebre ***k*** donde se ha encontrado el valor mínimo para la subcadenas indicada de acuerdo a la posición dentro de las matrices.

Implementación de la Solución

Se programó la función *MatrixChain(p)*, donde **p** es el arreglo de dimensiones de las matrices que se quieren multiplicar. Por ejemplo si tenemos dos matrices A y B, donde las dimensiones de A son a x b y las dimensiones de B son b x c, entonces *p = [a, b, c]*.
Por lo tanto la función *MatrixChain(p)* calcula todas las permutaciones de orden de multiplicación de las dimensiones de las matrices dadas en el arreglo p y almacena en la matriz **m** el valor mínimo de multiplicaciones. Una consideración a tomar es que se trabaja con índice inicial 1.

```
def MatrixChain(p):
  n=len(p)-1                                          %Establece el iterador
  s=[[0 for x in range(0, n+1)] for y in range(0,n+1)]  %Inicialización de matriz s en 0
  m=[[0 for x in range(0, n+1)] for y in range(0,n+1)]  %Inicialización de matriz m en 0
  for L in range(2, n+1):                             %FOR (L controla la longitud de la permutación)
    for i in range(1, n-L+2):                         %FOR  (i depende del valor de L)
      j = i+L-1                                       %
      m[i][j] = sys.maxsize                           % Inicializa en un valor max, espacio actual de matriz.
      for k in range(i, j-1+1):       # check all splits   %FOR k de i a j.
        print("L=%d   i=%d    j=%d    k=%d" % (L,i,j,k))  %
        q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]    % Prueba cada iteración dentro de la matriz
        if q < m[i][j]:                               %IF q es menor que m[i][j]
          m[i][j] = q                                 % Se guarda el valor menor
          s[i][j] = k                                 % Se guarda el valor k que dió el valor menor
  return m, s                                         %END IF, END FOR, END FOR, END FOR, RETURN
```

A continuación se muestra el resultado de la ejecución del programa tomando en cuenta las dimensiones de las matrices propuestas para la tarea:

$A_1 = 100 \times 4$
$A_2 = 4 \times 50$
$A_3 = 50 \times 20$

$A_4 = 20 \times 100$

De acuerdo a la convención previamente establecida tenemos que el valor del arreglo con las dimensiones de las matrices es *p = [100, 4, 50, 20, 100]*

Al ejecutar el *script* anterior obtenemos como resultado que el mejor caso es:

m=

0	**20000**	**12000**	**52000**
	0	**4000**	**12000**
		0	**100000**
			0

s =

0	**1**	**1**	**1**
	0	**2**	**3**
		0	**3**
			0

Es decir, el resultado óptimo es para este caso es:

(A1((A2A3)A4))

y el número de multiplicaciones es: **52000**

Por otro lado, para obtener el peor caso para este problema se realizaron las siguientes modificaciones al código, de tal forma que se almacene en m el número máximo de multiplicaciones obtenidas para cada subcadena. De la misma manera se almacena en la matriz s la posición k donde se encontró el máximo valor:

```
def MatrixChain(p):
n=len(p)-1                                  %Establece el tamaño del arreglo de dimensiones
m=[[0 for x in range(0, n+1)] for y in range(0,n+1)]   %Inicialización de matriz de óptimos en 0
s=[[0 for x in range(0, n+1)] for y in range(0,n+1)]   %Inicialización de matriz de posiciones en 0

for L in range(2, n+1):                     %FOR (L controla la longitud del producto de mtz)
        for i in range(1, n-L+2):           %FOR  (i depende del valor de L)
                j = i+L-1
                m[i][j] = -1                % Inicializa en un valor max, espacio actual de matriz.
                        for k in range(i, j-1+1):    %FOR k de i a j. Se evalúan todas las posiciones
                          q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]     % Calcula el costo de cálculo
                          if q >= m[i][j]:          %IF el costo es menor que el almacenado
                                m[i][j] = q         % Se actualiza el costo mínimo
                                s[i][j] = k         % Se guarda la posición
        return m, s                                 %END IF, END FOR, END FOR, END FOR, RETURN
```

Por lo tanto, al calcular cuál es el peor caso para el mismo arreglo de dimensiones ***p*** tenemos que:

m =

0	**20000**	**12000**	**620000**
	0	**4000**	**120000**
		0	**100000**
			0

s =

0	**1**	**2**	**2**
	0	**2**	**2**
		0	**3**
			0

Determinando que se requiere del mayor número de multiplicaciones cuando se realiza de la siguiente forma:

$$(A_1\ A_2)(A_3\ A_4)$$

requiriendo 620,000 multiplicaciones para calcular la matriz $A_{i..j}$

Referencias

[1] Shortest Addition Chains. Por David Wilson. http://wwwold.iit.cnr.it/staff/giovanni.resta/ac/

[2] Shortest Addition Chains. Por Achim F. http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html

[3] Menezes, A., Van Oorschot, P. y Vanstone, S. (1996). Handbook of Applied Cryptography. CRC Press.

[4] Cormen, T., Leiserson, C. E., etal. (2009). Introduction to Algorithms. MIT Press. Cambridge, Massachusetts.

Apendice A.

Códigos para calculo de exponentes.

```
#!/usr/bin/python
""" We are evaluating most efficient technique for
exponentiation"""

from time import clock
import time

#! BASIC TECHNIQUES FOR EXPONENTIATION
def parseBin(num):
    divs=num
    numB=[]
    temp=divs
    while (divs!=0):
        if((temp!=1)&((divs%2)==0)):
            numB.append(str(0))
        else:
            numB.append(str(1))
        temp=divs
        divs=int(divs/2)
    return numB[::-1]

def RLbinary(g,e):
    A=1
    S=g
    mults=0
    chain=list()
    expA=0
    expS=1
```

```
    while (e!=0):
        if ((e%2)!=0):
            A=A*S
            mults=mults+1
            expA+=expS
            chain.append(expA)

        e=int(e/2)

        if e!=0:
            S=S*S
            mults=mults+1
            expS+=expS
            chain.append(expS)
    return (A,mults,chain)

def LRbinary(g,e):
    A=1
    E=parseBin(e)
    mults=0
    chain=list()
    expA=0

    for i in range(len(E)):
        A=A*A
        mults=mults+1
        expA+=expA
        chain.append(expA)

        if (E[i]=='1'):
            A=A*g
            mults=mults+1
            expA+=1
            chain.append(expA)
    return (A,mults,chain)

def modularExp(a,b,n):
    c=0
    d=1
    bk=parseBin(b)
    temp=d
    mults=0
    chain=list()
    expA=0

    for i in range(len(bk)):
        c=2*c
        d=int((d*d)%n)
        temp=temp*temp
        mults=mults+1
```

```
        expA+=expA
        chain.append(expA)

        if(bk[i]=='1'):
            c=c+1
            d=int((d*a)%(n))
            temp=temp*a
            mults=mults+1
            expA+=1
            chain.append(expA)
    #return d
    return (temp,mults,chain)


# 2,77
print("\n**************** EXP(2,77) *****************\n")

print("\n\n LR BINARY")
startF=int(round(time.time() * 1000000))
exponts,mults,chain=LRbinary(2,77)
endF=int(round(time.time() * 1000000))
print("TIME: ", (endF-startF), "microseconds")
print("Result:",exponts)
print("Number of mult =",mults)
print("Add Chain = ",chain)


print("\n RL BINARY")
startF=int(round(time.time() * 1000000))
exponts2,mults3,chain=RLbinary(2,77)
endF=int(round(time.time() * 1000000))
print("RLbinary ",exponts2)
print("TIME: ", endF-startF, "microseconds")
print("mults ",mults3)
print("Add Chain = ",chain)


print("\n Modular Exponentiation")
startF=int(round(time.time() * 1000000))
expon,mults1,chain=modularExp(2,77,79)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print ("Result:", expon)
print ("Number of mult =", mults1)
print("Add Chain = ",chain)


# 2,195
print("\n\n**************** EXP(2,195) *****************\n")
```

```
print("\n RL BINARY")
startF=int(round(time.time() * 1000000))
exponts2,mults3,chain=RLbinary(2,195)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print("Result:",exponts2)
print("Number of mult =",mults3)
print("Add Chain = ",chain)

print("\n Modular Exponentiation")
startF=int(round(time.time() * 1000000))
expon,mults1,chain=modularExp(2,195,196)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print ("Result:", expon)
print ("Number of mult =", mults1)
print("Add Chain = ",chain)

print("\n\n LR BINARY")
startF=int(round(time.time() * 1000000))
exponts,mults,chain=LRbinary(2,195)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print("Result:",exponts)
print("Number of mult =",mults)
print("Add Chain = ",chain)



# 2,631
print("\n\n***************** EXP(2,631) ******************\n")
print("\n Modular Exponentiation")
startF=int(round(time.time() * 1000000))
expon,mults1,chain=modularExp(2,631,632)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print ("Result:", expon)
print ("Number of mult =", mults1)
print("Add Chain = ",chain)

print("\n\n LR BINARY")
startF=int(round(time.time() * 1000000))
exponts,mults,chain=LRbinary(2,631)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print("Result:",exponts)
print("Number of mult =",mults)
print("Add Chain = ",chain)

print("\n RL BINARY")
```

```
startF=int(round(time.time() * 1000000))
exponts2,mults3,chain=RLbinary(2,631)
endF=int(round(time.time() * 1000000))
print("TIME: ", endF-startF, "microseconds")
print("Result:",exponts2)
print("Number of mult =",mults3)
print("Add Chain = ",chain)
```

Código para cálculo de multiplicaciones mínimas de matrices.

```
#This program solves the chain multiplication problem

import sys
def MatrixChain(p):
  n=len(p)-1
  s=[[0 for x in range(0, n+1)] for y in range(0,n+1)]
  m=[[0 for x in range(0, n+1)] for y in range(0,n+1)]
  for i in range(1,n+1):
    m[i][i]=0
  for L in range(2, n+1):
    for i in range(1, n-L+2):
      j = i+L-1
      m[i][j] = sys.maxsize
      for k in range(i, j-1+1):        # check all splits
        print("L=%d   i=%d    j=%d    k=%d" % (L,i,j,k))
        q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
        if q < m[i][j]:
          m[i][j] = q
          s[i][j] = k

  return m, s

def printParens(s,i,j):
  if i==j:
    print("A%d"%i,end="")
  else:
    print ("(",end="")
    printParens(s,i,s[i][j])
    printParens(s,s[i][j]+1,j)
    print (")",end=""),

p1=[30,35,15,5,10,20,25]
p1=[100,4,50,20,100]
m,s=MatrixChain(p1)
n=len(p1)

printParens(s, 1, n-1)
print("")
print("El numero de multiplicaciones es: %d"%m[1][n-1])
```