3 Diseño de Algoritmos Eficientes

En este capítulo veremos un conjunto de ideas que permiten diseñar algoritmos que, en muchos casos, son de los más eficientes que se conocen para sus respectivos problemas.

Dividir para Reinar

Este es un método de diseño de algoritmos que se basa en subdividir el problema en sub-problemas, resolverlos recursivamente, y luego combinar las soluciones de los sub-problemas para construir la solución del problema original. Es necesario que los subproblemas tengan la misma estructura que el problema original, de modo que se pueda aplicar la recursividad.

Ejemplo: Multiplicación de Polinomios

Supongamos que tenemos dos polinomios $A(x)$ y $B(x)$, cada uno de grado $n-1$:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1}$$

representados por sus respectivos arreglos de coeficientes $a[0], \ldots, a[n-1]$ y $b[0], \ldots, b[n-1]$.

El problema consiste en calcular los coeficientes $c[0], \ldots, c[2n-2]$ del polinomio producto $C(x) = A(x)B(x)$.

La manera obvia de resolver este problema es multiplicando cada término de $A(x)$ por cada término de $B(x)$ y acumulando los resultados que corresponden a la misma potencia de x:

In [2]:

```
import numpy as np
def multpol(a, b):
    n=len(a)
    assert len(b)==n
    c=np.zeros(2*n-1)
    for i in range(0,n):
        for j in range(0,n):
            c[i+j]+=a[i]*b[j]
    return c
```

In [6]:

```
multpol(np.array([2,3,-6,1]), np.array([1,-1,3,1]))
```

Out[6]:

```
array([  2.,   1.,  -3.,  18., -16.,  -3.,   1.])
```

Evidentemente, este algoritmo demora tiempo $O(n^2)$. ¿Es posible hacerlo más rápido? Para esto, aplicaremos la técnica de *dividir para reinar*.

Supongamos que n es par, y dividamos los polinomios en dos partes, separando las potencias bajas de las altas. Por ejemplo, si

$$A(x) = 2 + 3x - 6x^2 + x^3$$

lo podemos reescribir como

$$A(x) = (2 + 3x) + (-6 + 3x)x^2$$

En general, podemos reescribir $A(x)$ y $B(x)$ como

$$A(x) = A'(x) + A''(x)x^{n/2}$$
$$B(x) = B'(x) + B''(x)x^{n/2}$$

y entonces (omitiendo los "(x)" para simplificar la notación),

$$C = A'B' + (A'B'' + A''B')x^{n/2} + A''B''x^n$$

Esto se puede implementar con 4 multiplicaciones recursivas, cada una involucrando polinomios de la mitad del tamaño. Nótese que las multiplicaciones por potencias de x son solo realineaciones de los arreglos de coeficientes, de modo que son "gratis".

Si llamamos $T(n)$ al número total de operaciones, éste obedece la ecuación de recurrencia

$$T(n) = 4T\left(\frac{n}{2}\right) + Kn$$

para alguna constante K.

Por el Teorema Maestro, con $p = 4$, $q = 2$, $r = 1$, tenemos

$$T(n) = O(n^2)$$

lo cual no es mejor que el algoritmo anterior.

Afortunadamente, hay una manera de obtener un algoritmo realmente más eficiente. Si calculamos

$$D = (A' + A'')(B' + B'')$$
$$E = A'B'$$
$$F = A''B''$$

podemos construir el polinomio C de la manera siguiente:

$$C = E + (D - E - F)x^{n/2} + Fx^n$$

¡lo cual utiliza solo 3 multiplicaciones recursivas!

Usando nuevamente el Teorema Maestro, esta vez con $p = 3$, tenemos que

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$$

Programación Dinámica

Hay ocasiones en que la simple aplicación de la recursividad conduce a algoritmos muy ineficientes, pero es posible evitar esa ineficiencia con un uso adecuado de memoria.

Ejemplo: Cálculo de un número de Fibonacci

Recordemos la ecuación de Fibonacci

$$
\begin{aligned}
f_n &= f_{n-1} + f_{n-2} \text{ para } n \geq 2 \\
f_0 &= 0 \\
f_1 &= 1
\end{aligned}
$$

algunos de cuyos valores son:

n	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
f_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

Queremos resolver el siguiente problema: dado un n, calcular f_n.

A partir de la ecuación de recurrencia podemos escribir de inmediato una solución recursiva:

In [8]:

```
def fibonacci(n):
    if n<=1:
        return n
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

In [9]:

```
print(fibonacci(10))
```

```
55
```

El problema es que, para n grande, este algoritmo es horriblemente ineficiente. El motivo de esto es que, a medida que se van ejecutando las llamadas recursivas, un mismo número de Fibonacci puede calcularse múltiples veces, independientemente de si ya se ha calculado antes.

Una forma de dimensionar esta ineficiencia es calcular, por ejemplo, el número de operaciones de suma que se hacen al calcular `fibonacci(n)`. Llamemos s_n a ese número de sumas. Es fácil ver que

$$\begin{aligned} s_n &= s_{n-1} + s_{n-2} + 1 \text{ para } n \geq 2 \\ s_0 &= 0 \\ s_1 &= 0 \end{aligned}$$

Si definimos una nueva función incógnita $t_n = s_n + 1$, tenemos que

$$\begin{aligned} t_n &= t_{n-1} + t_{n-2} \text{ para } n \geq 2 \\ t_0 &= 1 \\ t_1 &= 1 \end{aligned}$$

Esta es la misma ecuación de Fibonacci, comenzando un paso más adelante, y por lo tanto su solución es $t_n = f_{n+1}$, y el número de sumas es $s_n = f_{n+1} - 1 = \Theta(\phi^n)$. Conclusión: ¡el tiempo que demora la ejecución de `fibonacci(n)` crece exponencialmente!

Evitando la ineficiencia: Memoización

La ineficiencia de la solución recursiva se debe, como dijimos antes, a que un mismo valor de la función f se calcula y recalcula múltiples veces. Una forma de evitar esto es incorporar una memoria auxiliar, en forma de un arreglo F inicializado con ceros. La primera vez que se pide calcular un f_n dado, lo hacemos recursivamente, pero dejamos el valor anotado en $F[n]$. Las siguientes veces lo tomamos del arreglo, sin incurrir de nuevo en el costo del cálculo recursivo:

In [14]:

```
def fibonacci(n):
    F=np.zeros(n+1)
    def fib_rec(k):
        if k>0 and F[k]==0: # primera vez que se calcula
            if k<=1:
                F[k]=k
            else:
                F[k]=fib_rec(k-1)+fib_rec(k-2)
        return F[k]
    return fib_rec(n)
```

In [15]:

```
print(fibonacci(10))
```

```
55.0
```

La introducción de esta memoria auxiliar tiene como efecto transformar el algoritmo original, de tiempo exponencial, en un algoritmo de tiempo lineal $\Theta(n)$.

Evitando la ineficiencia: Tabulación

La técnica de *memoización* va llenando el arreglo auxiliar F a medida que sus valores son solicitados. Este método es bastante general, pero se puede mejorar si logramos encontrar un orden para ir llenando el arreglo F que garantice que cuando se requiere el valor de un cierto casillero, éste ya está llenado.

En el caso de Fibonacci, esto se logra simplemente al ir llenando los casilleros $F[k]$ en orden creciente de k. Esta técnica se llama *tabulación*:

In [18]:

```
def fibonacci(n):
    F=np.zeros(n+1)
    F[0]=0
    F[1]=1
    for k in range(2,n+1):
        F[k]=F[k-1]+F[k-2]
    return F[n]
```

In [19]:

```
print(fibonacci(10))
```

```
55.0
```

Es evidente que el tiempo que demora este algoritmo es $\Theta(n)$.

Derrotando al algoritmo lineal

Si bien parecería que para calcular f_n es necesario calcular previamente todos los f_k, para $0 \leq k < n$, en realidad esto no es cierto.

Introduzcamos una función incógnita adicional g_n en la ecuación de Fibonacci, definiéndola como $g_n = f_{n-1}$. La ecuación puede reescribirse así:

$$\begin{aligned} f_n &= f_{n-1} + g_{n-1} \\ g_n &= f_{n-1} \\ f_1 &= 1 \\ g_1 &= 0 \end{aligned}$$

Esto se puede reescribir en forma matricial:

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ g_{n-1} \end{pmatrix} \text{ para } n \geq 2$$

con la condición inicial

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Este ecuación es muy simple de resolver "desenrrollándola", y su solución es

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Por lo tanto, para resolver el problema del calcular f_n, basta evaluar esta fórmula matricial y luego tomar la primera componente del vector resultante.

Recordando que para la elevación a potencia podemos usar el algoritmo binario, la evaluación de la fórmula se puede hacer en tiempo logarítmico, y por lo tanto el problema de calcular f_n se puede resolver en tiempo $\Theta(\log n)$.

La siguiente es una versión del algoritmo `potencia` adaptada para calcular $B = A^n$ cuando A es una matriz cuadrada:

In [34]:

```
def potencia(A, n):
    B=np.eye(len(A)) # matriz identidad
    k=n
    C=A
    while k>0:
        while k%2==0:
            C=np.dot(C,C) # C=C**2
            k//=2
        B=np.dot(B,C) # B=C*C
        k-=1
    return B
```

In [35]:

```
def fibonacci(n):
    F=np.dot(potencia(np.array([[1,1],[1,0]]),n-1), np.array([[1],[0]]))
    return F[0,0]
```

In [36]:

```
print(fibonacci(10))
```

```
55.0
```

Ejemplo: Encontrar la parentización óptima para multiplicación de n matrices

Hemos visto que las técnicas de *memoización* y de *tabulación* nos permiten construir algoritmos eficientes en algunos problemas en que la recursividad aplicada directamente daría soluciones muy ineficientes.

Cuando estas técnicas se aplican a problemas de optimización, hablamos de *programación dinámica*.

Consideremos el siguiente problema: Dadas tres matrices A, B y C para las cuales se desea calcular su producto ABC, ¿qué es más eficiente, calcular $(AB)C$ o calcular $A(BC)$?

La respuesta depende de las dimensiones de las matrices involucradas. Si una matriz A es de $p \times q$ y otra matriz B es de $q \times r$, calcular su producto AB utilizando el algoritmo usual requiere hacer pqr multiplicaciones escalares, y un número similar de sumas.

Para nuestro problema de calcular ABC, supongamos por ejemplo que A es de 100×10, B de 10×100 y C de 100×10, tenemos que

- Calcular $(AB)C$ requiere $100 \times 10 \times 100 + 100 \times 100 \times 10 = 200.000$ multiplicaciones
- Calcular $A(BC)$ requiere $10 \times 100 \times 10 + 100 \times 10 \times 10 = 20.000$ multiplicaciones

La respuesta, por lo tanto, es que para las dimensiones dadas, la parentización óptima es $A(BC)$.

Consideremos ahora el problema general. Dadas n matrices $A_1, A_2, \ldots, A_n$ y números $p[0], p[1], \ldots, p[n]$ tales que la matriz A_i es de $p[i-1] \times p[i]$, encontrar el costo (en número de multiplicaciones) de la parentización óptima para calcular el producto

$$A_1 A_2 \cdots A_n$$

Generalicemos el problema para poder abordarlo recursivamente (o inductivamente). Supongamos que el problema es encontrar el costo de la parentización óptima para calcular el producto

$$A_i \cdots A_j$$

para $1 \leq i \leq j \leq n$. Llamemos $m[i, j]$ a este costo óptimo.

En el caso $i = j$ el producto involucra a una sola matriz, así que, trivialmente, $m[i, i] = 0$. En el caso $i < j$, supongamos que parentizamos de modo que el producto se factorice como

$$(A_i \cdots A_k)(A_{k+1} \cdots A_j)$$

para algún $k \in [i \,..\, j-1]$. Suponiendo que cada producto parentizado se ha calculado en forma óptima, el costo total sería

$$m[i, k] + m[k+1, j] + p[i-1] \times p[k] \times p[j]$$

Esto no es necesariamente óptimo para el producto $A_i \cdots A_j$, porque podríamos haber elegido el valor equivocado de k. Para asegurarnos de alcanzar el óptimo, tenemos que minimizar sobre todo k:

$$m[i, j] = \min_{i \leq k \leq j-1} \{m[i, k] + m[k+1, j] + p[i-1] \times p[k] \times p[j]\}$$

Esto lo podríamos implementar mediante una función recursiva, pero, tal como sucedía en el ejemplo de Fibonacci, ésta demoraría un tiempo exponencial en ejecutarse, porque generaría y evaluaría todas las parentizaciones posibles. Esta búsqueda exhaustiva es un método que encuentra la respuesta correcta (se le suele llamar "método fuerza bruta"), pero usualmente es demasiado ineficiente.

Afortunadamente, en este caso podemos usar tabulación, porque hay solo alrededor de $n^2/2$ casilleros que llenar en la matriz m, y los podemos ir llenando en un orden tal que al calcular el mínimo sobre todo k, los casilleros necesarios ya han sido llenados previamente.

En efecto, introduzcamos una nueva variable $d = j - i + 1$. Esto es el número de matrices involucradas en el producto $A_i \cdots A_j$. Lo que haremos será ir llenando la matriz en orden ascendente de la variable d, comenzando con el caso trivial $d = 1$, hasta terminar con el caso $d = n$, que corresponde a la solución del problema original.

In [43]:

```
import math
def opti_multi_mat(p):
    n=len(p)-1
    m=np.zeros((n+1,n+1)) # Esto ya deja la diagonal en cero
    for d in range(2,n+1):
        for i in range(1,n-d+2):
            j=i+d-1
            m[i,j]=math.inf # +infinito
            for k in range(i,j):
                q=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j]
                if q<m[i,j]:
                    m[i,j]=q
    return m[1,n]
```

In [44]:

```
p=np.array([100,10,100,10])
print(opti_multi_mat(p))
```

```
20000.0
```

Es fácil ver que este algoritmo corre en tiempo $O(n^3)$, porque debe rellenar $\Theta(n^2)$ casilleros, y cada uno puede requerir examinar n valores posibles de k en el peor caso. Esto es significativamente mejor que el algoritmo de fuerza bruta.

Esta cota cúbica podría, sin embargo, ser un poco exagerada, porque en muchos casos la variable k toma mucho menos que n valores. Un cálculo más preciso nos señala que el número total de veces que se ejecuta el cuerpo del ciclo `for k` es igual a

$$\sum_{1 \leq i \leq k < j \leq n} 1 = \frac{n(n-1)(n+1)}{6} = \Theta(n^3)$$

Por lo tanto el algoritmo en realidad demora un tiempo cúbico.

Como vemos, el resultado del proceso es el costo óptimo, pero eso no nos da ninguna información sobre cuál es la parentización óptima. Pero en realidad la información está ahí, porque el valor de k para el cual se alcanza el mínimo nos dice dónde separar la parentización en cada caso. Basta entonces con que dejemos anotado, para cada i, j cuál es el valor de k para el que se alcanza el mínimo. Llamemos $s[i, j]$ a ese valor de k.

Modifiquemos nuestra función para que construya y retorne la matriz s además del costo óptimo, y escribamos otra función que, dada esa matriz s, imprima la fórmula parentizada de la manera óptima.

In [45]:

```
import math
def opti_multi_mat(p):
    n=len(p)-1
    m=np.zeros((n+1,n+1)) # Esto ya deja la diagonal en cero
    s=np.zeros((n+1,n+1))
    for d in range(2,n+1):
        for i in range(1,n-d+2):
            j=i+d-1
            m[i,j]=math.inf # +infinito
            for k in range(i,j):
                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 # anotamos dónde se alcanza el min
    return (m[1,n],s)
```

In [60]:

```
def paren_desde_hasta(s,i,j):
    if i==j:
        return "A"+str(i)
    else:
        return "("+paren_desde_hasta(s,i,int(s[i,j]))+" "\
               +paren_desde_hasta(s,int(s[i,j])+1,j)+")"

def parentizacion(s):
    n=len(s)-1
    return paren_desde_hasta(s,1,n)
```

In [62]:

```
p=np.array([100,10,100,10])
(opt,s)=opti_multi_mat(p)
print(parentizacion(s))
print("Costo=", opt)
```

```
(A1 (A2 A3))
Costo= 20000.0
```

Para recapitular, la técnica de diseño de programación dinámica divide un problema en varios subproblemas con la misma estructura que el problema original, luego se resuelven dichos subproblemas y finalmente, a partir de éstos, se obtiene la solución al problema original. La diferencia radica en que la programación dinámica se ocupa cuando los subproblemas se repiten, como en el cálculo de los números de Fibonacci. En este caso, en vez de usar recursión para obtener las soluciones a los subproblemas éstas se van tabulando en forma bottom-up, y luego estos resultados son utilizados para resolver subproblemas más grandes. De esta forma, se evita el tener que realizar el mismo llamado recursivo varias veces.

La programación dinámica se ocupa en general para resolver problemas de optimización (maximización o minimización de alguna función objetivo). Estos problemas pueden tener una o varias soluciones óptimas, y el objetivo es encontrar alguna de ellas. Los pasos generales para utilizar programación dinámica en la resolución de un problema son los siguientes:

- Encontrar la subestructura óptima del problema, es decir, encontrar aquellos subproblemas en los que se compone el problema original, tal que si uno encuentra sus soluciones óptimas entonces es posible obtener la solución óptima al problema original.
- Definir el valor de la solución óptima en forma recursiva.
- Calcular el valor de la solución partiendo primero por los subproblemas más pequeños y tabulándo las soluciones, lo que luego permite obtener la solución de subproblemas más grandes. Terminar cuando se tiene la solución al problema original.

Estos pasos permiten obtener el valor óptimo de la solución al problema. También es posible ir guardando información extra en cada paso del algoritmo, que luego permita reconstruir el camino realizado para hallar la solución óptima (por ejemplo, para obtener la instancia específica de la solución óptima, y no sólo el valor óptimo de la función objetivo).