Tarea

Nombres: Mario Becerra, Miguel Vilchis
Fecha: Septiembre de 2016

1. Descomposición de suma en enteros

Usar un algoritmo que encuentre la forma óptima de descomponer la suma en enteros. Usarlo para calcular las siguientes potencias con 2 espacios en memoria y n espacios en memoria:

- x^{77}
- x^{511}
- x^{3631}

El algoritmo que utilizamos es el siguiente:
Sea x la base y e el exponente exponente al que queremos elevar la base entonces:

Algorithm 1 Algoritmo para calcular la potencia de un número con 2 espacios de memoria

Require: Two nonnegative integers x and e
  $a = 1$
  $b = x$
  while $e! = 0$ **do**
    if $e\,\%2! = 0$ **then**
      $a = a \times b$
    end if
      $e = int(\frac{e}{2})$
    if $e! = 0$ **then**
      $b = b * b$
    end if
  end while

Calculo de la complejidad
La complejidad del algoritmo descrito depende de la representación binaria del exponente, ya que, por cada iteración se hace un recorrido a la derecha de la representación binaria del exponente, que es consecuencia de la división del exponente entre 2. Es claro ver que el número de iteraciones totales del while es $log(e)$ Con base en lo descrito tenemos que la complejidad del algoritmo es de :

$$O((elog(x))^k)$$

Aplicando este algoritmo a nuestro problema obtenemos lo siguiente:

x^{77} Número de operaciones: 10

M_1	M_2
8 $x^0 \times x^1(x^1)$	
	$x^1 \times x^1(x^2)$
	$x^2 \times x^2(x^4)$
$x^1 \times x^4(x^5)$	
	$x^4 \times x^4(x^8)$
$x^5 \times x^8(x^{13})$	
	$x^8 \times x^8(x^{16})$
	$x^{16} \times x^{16}(x^{32})$
	$x^{32} \times x^{32}(x^{64})$
$x^{13} \times x^{64}(x^{77})$	

x^{511} Número de operaciones: 17

M_1	M_2
$x^0 \times x^1(x^1)$	
	$x^1 \times x^1(x^2)$
$x^1 \times x^2(x^3)$	
	$x^2 \times x^2(x^4)$
$x^3 \times x^4(x^7)$	
	$x^4 \times x^4(x^8)$
$x^7 \times x^8(x^{15})$	
	$x^8 \times x^8(x^{16})$
$x^{15} \times x^{16}(x^{31})$	
	$x^{16} \times x^{16}(x^{32})$
$x^{31} \times x^{32}(x^{63})$	
	$x^{32} \times x^{32}(x^{64})$
$x^{63} \times x^{64}(x^{127})$	
	$x^{64} \times x^{64}(x^{128})$
$x^{127} \times x^{128}(x^{255})$	
	$x^{128} \times x^{128}(x^{256})$
$x^{255} \times x^{256}(x^{511})$	

x^{3631} Número de operaciones: 19

M_1	M_2
$x^0 \times x^1(x^1)$	
	$x^1 \times x^1(x^2)$
$x^1 \times x^2(x^3)$	
	$x^2 \times x^2(x^4)$
$x^3 \times x^4(x^7)$	
	$x^4 \times x^4(x^8)$
$x^7 \times x^8(x^{15})$	
	$x^8 \times x^8(x^{16})$
	$x^{16} \times x^{16}(x^{32})$
$x^{15} \times x^{32}(x^{47})$	
	$x^{32} \times x^{32}(x^{64})$
	$x^{64} \times x^{64}(x^{128})$
	$x^{128} \times x^{128}(x^{256})$
	$x^{256} \times x^{256}(x^{512})$
$x^{47} \times x^{512}(x^{559})$	
	$x^{512} \times x^{512}(x^{1024})$
$x^{559} \times x^{1024}(x^{1583})$	
	$x^{1024} \times x^{1024}(x^{2048})$
$x^{1583} \times x^{2048}(x^{3631})$	

Note que el anterior algoritmo genera una partición de los número naturales, del lado izquierdo trabajamos con exponentes impares y del lado derecho con exponentes pares. Ahora para generalizar el algoritmo, aplicamos el principio de divisibilidad de los enteros generando n clases de equivalencia o particiones para n espacios de memoria. Sea e_n la representación de e en base n, tenemos el siguiente algoritmo:

Algorithm 2 Algoritmo para calcular la potencia de un número con n espacios de memoria

Require: Two nonnegative integers x , e and n

```
espacios = [x^i for i in range(n)]
r = 1
for digiti in e_n do
r = r * r
    if digit then
     r = r × espacios[digit]
    end if

end for
```

2. Multiplicación de matrices

Dadas n matrices compatibles, encontrar el mejor ordenamiento para multiplicarlas de tal forma que se minimice el número de multiplicaciones requeridas para calcular el producto, o sea, se quiere calcular $A_1A_2 \dots A_n$ en el orden que minimice el número de multiplicaciones de escalares necesarias. Por ejemplo, si $n = 3$, los paréntesis de las matrices se pueden acomodar de distintas formas para llevar a cabo el producto $(A_1A_2)A_3$ o $A_2(A_2A_3)$.

De hecho, el número de formas de acomodar los paréntesis $P(n)$ de una cadena de matrices de tamaño n está dado por:

$$P(n) = \begin{cases} 1 & \text{si } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

Esta función tiene una complejidad $\Omega(4^n/n^{3/2})$. Es decir, un método por fuerza bruta (calcular todas las posibles formas de acomodar entre paréntesis) tomaría demasiado tiempo en encontrar el acomodo óptimo. Notar que esto es solo para saber el acomodo óptimo, no se está haciendo ninguna multiplicación aún.

Una alternativa para resolver este problema es utilizar programación dinámica, técnica utilizada en muchas áreas para optimizar una función objetivo sujeta a distintas restricciones. En general, la programación dinámica sirve bien cuando se puede dividir el problema en distintos subproblemas que se traslapan. La programación dinámica se puede enunciar en cuatro pasos:

1. Caracterizar la estrucutra de la solución óptima.
2. Definir recursivamente el valor de la solución óptima.
3. Calcular el valor de la solución óptima, típicamente, *de abajo hacia arriba*.
4. Construir la solución óptima a partir de la información calculada.

Paso 1

Sea $A_{i\ldots j} = A_i A_{i+1} \ldots A_j$, con $j \geq i$. Si $j > i$ estrictamente, entonces para acomodar entre paréntesis el producto $A_i A_{i+1} \ldots A_j$, se deben calcular los productos $A_{i\ldots k}$ y $A_{k+1\ldots j}$ para algún entero k tal que $i \leq k < i$, para después calcular el producto $A_{i\ldots k} A_{k+1\ldots j}$ y obtener $A_{i\ldots j}$. El costo de acomodar los paréntesis de esta forma es el costo de calcular $A_{i\ldots k}$, de calcular $A_{k+1\ldots j}$ y de calcular su producto.

Cualquier solución a una instancia no trivial (i.e. $j > i$) del problema de la multiplicación secuencial de matrices requiere que se divida el producto en dos, y cualquier solución óptima contiene dentro de ella misma soluciones óptimas a instancias de subproblemas. Así, se puede construir una solución óptima a una instancia del problema al dividir el problema en dos subproblemas (encontrar el acomodo óptimo de paréntesis de $A_{i\ldots k}$ y $A_{k+1\ldots j}$), encontrar soluciones óptimas para estos subproblemas y combinar estas soluciones de subproblemas. Cuando se esté buscando la posición correcta para dividir el producto, se debe de estar seguro de que se hayan considerado todos los lugares posibles, para así estar seguros de que ese está analizando el óptimo.

Paso 2

Aquí se define recursivamente el costo de una solución óptima en términos de soluciones óptimas a subproblemas. Para el problema de la multiplicación secuencial de matrices, un subproblema se define como el problema de determinar el acomodo de paréntesis que minimiza el costo de multiplicar $A_i A_{i+1} \ldots A_j$ para $1 \leq i \leq j \leq n$. Sea $m(i, j)$ el número mínimo de multiplicaciones escalares necesarias para calcular el producto $A_{i\ldots j}$.

Se puede definir la función $m(i, j)$ recursivamente como

$$m(i,j) = \begin{cases} 0 & \text{si } i = j \\ \min_{k, i \leq k < j} \{m(i,k) + m(k+1,j) + p_{i-1} p_k p_j\} & \text{si } i < j \end{cases} \tag{1}$$

donde $p = [p_0, p_1, p_2, ..., p_n]$ es un vector que contiene todas las dimensiones únicas de las matrices en el orden de la cadena de matrices, es decir, cada matriz A_i es de dimensión $p_{i-1} \times p_i$. Esto todavía no da la información necesaria para construir una solución óptima, para eso, se define $s(i, j)$ el valor de k en el que se tiene el acomodo óptimo de paréntesis en $A_i A_{i+1} \ldots A_j$. Es decir, $s(i, j)$ es el valor de k tal que $m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j$.

Paso 3

Con la recurrencia de (1), se podría construir un algoritmo recursivo que encuentre el costo mínimo $m(1, n)$ de multiplicar $A - 1 \ldots A_n$, pero este algoritmo tomaría un tiempo exponencial. Sin embargo, se puede notar que hay relativamente pocos subproblemas distintos: un subproblema para cada i y j que satisfacen $1 \leq i \leq j \leq n$, esto es $\binom{n}{2} + 2$. Un algoritmo recursivo se encontraría varias veces los mismos problemas muchas veces en distintas ramas del árbol de recursión.

Es aquí donde entra la programación dinámica, la cual evita que se estén calculando muchas veces los mismos problemas. El primer paso es construir una tabla que ayude a calcular las soluciones. Se necesita una matriz bidimensional $m[i, j]$. Ahora, en la función recursiva $m(i, j)$ se va buscando la solución a partir de las soluciones más básicas. Entonces, usando un esquema *bottom-up*, se calculan las soluciones de cadenas tamaño 0, es decir $i = j$. Luego de abajo hacia arriba se va calculando la solución. Se necesita una matriz adicional s en la cual se van a guardar las entradas $s(i, j)$, o sea, qué índices de k van teniendo el costo óptimo de calcular $m(i, j)$. Al final, con la tabla s se va a construir la solución óptima.

El algoritmo de programación dinámica que encuentra la solución de $m(i, j)$ es el siguiente. El algoritmo recibe el vector p con las dimensiones de las matrices.

Require: $n = length(p) - 1$
Ensure: $m[1...n, 1...n]$ & $s[1...n, 1...n]$
  for $i = 1...n$ **do**
    $m[i, i] = 0$
  end for
  for $l = 2...n$ **do**
    for $i = 1...n - l + 1$ **do**
      $j = i + l - 1$
      $m[i, j] = \infty$
        for $k = i...j - 1$ **do**
          $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
          if $q < m[i, j]$ **then**
            $m[i, j] = q$
            $s[i, j] = k$
          end if
        end for
    end for
  end for

Al final, el algoritmo regresa m y s. Este algoritmo corre en $\Omega(n^3)$ y requiere de $\Theta(n^2)$ de espacio para almacenar las tablas m y s, por lo que se puede decir que es mucho más eficiente que el algoritmo de fuerza bruta.

Paso 4

La información que se necesita para tener la solución óptima está en la tabla s. Cada entrada $s(i, j)$ tiene un valor de k tal que una forma óptima de acomodar los paréntesis del producto $A_i \dots A_j$ separa el producto entre A_k y A_{k+1}. De esta forma se sabe que la multiplicación final debe ser $A_{1...s(1,n)} A_{s(1,n)+1...n}$. El siguiente procedimiento recursivo, llamado `Imprime_parentesis()`,imprime la forma óptima de acomodar los paréntesis en el producto.

Require: s, i, j
  if $i == j$ **then**
    print $'A'_i$
  end if
  else
    print '('
    `Imprime_parentesis`$(s,\ i,\ s(i, j))$
    `Imprime_parentesis`$(s,\ s(i, j) + 1,\ j)$
    print ')'
  end else

Para ver la forma óptima de poner los paréntesis, una tendría que llamar la función `Imprime_parentesis`$(s, 1, n)$.

Implementación

Se implementaron los doa algoritmos mencionados en R y en C++, y se probaron con el siguiente conjunto de matrices:

- $A_1 \in \mathbb{R}^{100\times 4}$
- $A_2 \in \mathbb{R}^{4\times 50}$
- $A_3 \in \mathbb{R}^{50\times 20}$
- $A_4 \in \mathbb{R}^{20\times 100}$

El resultado final fue que se multiplicaran en el orden $A_1 \times ((A_2 \times A_3) \times A_4)$ con un total de 52,000 multiplicaciones de escalares.

El código está disponible en `https://github.com/mariobecerra/Analisis_Algoritmos_Tarea_05`.