Tarea 3: Una muestra de problemas

Tania Patiño 137175
Víctor Peña 119413
Javier Sagastuy 122627
Ernesto Valdés 114049

Análisis de Algoritmos
Doctor Osvaldo Gabriel Cairó Battistutti
ITAM
Otoño 2013
México, Distrito Federal

***Abstract*— El análisis de algoritmos no puede disociarse del estudio de problemas. Esta tarea consistió de dos problemas que pretenden introducir al alumno al tipo de problemas que se abordan en esta área de ciencias de la computación.**

I. POTENCIACIÓN

En este ejercicio se busca resolver el problema de cómputo de X^n para potencias muy grandes. Sabemos que llevar a cabo los cálculos por fuerza bruta es muy costoso y nada eficiente. Es por está razón que la idea es encontrar la secuencia de multiplicaciones que optimiza la memoria que una computadora utiliza para realizar este cálculo.

En este ejercicio derivamos una manera de llevar a cabo la operación de potenciación para cada una de las X elevadas a las siguientes potencias 77, 195 y 631. Hicimos primero las operaciones usando *dos localidades* de memoria y después lo hicimos para *n localidades*. Con base en los patrones que vimos al resolver estos tres casos particulares, intentamos obtener algoritmos generales para resolver el problema con cada una de las restricciones de memoria.

A. Ejemplos

1) *Usando dos localidades de memoria:*
   a) *n = 77*

X								
X^2	X^4	X^8	X^9	X^{18}	X^{19}	X^{38}	X^{76}	X^{77}

Resultado = 9 multiplicaciones

b) n = 195

X									
X^2	X^3	X^5	X^{12}	X^{24}	X^{48}	X^{96}	X^{97}	X^{144}	X^{195}

Resultado = 10 multiplicaciones

c) n = 631

X														
X^2	X^4	X^8	X^9	X^{18}	X^{19}	X^{38}	X^{39}	X^{98}	X^{156}	X^{157}	X^{314}	X^{315}	X^{630}	X^{631}

Resultado = 15 multiplicaciones

*2) Usando **m** localidades de memoria:*

a) n = 77

X	X^2	X^3
X^5	X^7	X^{14}
X^{21}	X^{28}	X^{56}
X^{77}		

Resultado = 9 multiplicaciones

b) n = 195

X	X^2	X^3
X^6	X^{12}	X^{21}
X^{48}	X^{96}	X^{192}
X^{195}		

Resultado = 9 multiplicaciones

c) n = 631

X	X^2	X^3	X^5
X^7	X^{14}	X^{20}	X^{35}
X^{70}	X^{140}	X^{280}	X^{560}
X^{630}	X^{631}		

Resultado = 13 multiplicaciones

B. Algoritmos

1) Usando dos localidades de memoria:

Analizando el caso resuelto en clase para n=15 y la solución que obtuvimos para n=77, pudimos darnos cuenta de un punto importante: Cuando se tienen dos localidades de memoria, de acuerdo a las reglas vistas en clase, una de las localidades debe contener siempre el valor X, el número que se desea elevar a cierta potencia. Con esta restricción, en realidad sólo podemos operar sobre la otra localidad de memoria. Esto restringe las operaciones posibles a dos. Podemos multiplicar el valor de la segunda localidad por si mismo, resultando en duplicar su exponente o podemos multiplicar el valor de la segunda localidad por el de la primera localidad. Esto último equivale a incrementar el exponente del valor en la segunda localidad en 1. Nótese que la segunda localidad siempre se inicia con X^2 que se obtiene de multiplicar X por sí mismo.

Así, decidimos implementar el algoritmo en "reversa" aplicando las funciones inversas a las descritas anteriormente. En resumen: se inicia con el exponente al que se desea elevar X, n. Si es un número impar se le resta 1. Si es un número par se divide entre 2. Al resultado se le vuelve a aplicar el algoritmo. Esto se hace hasta que el valor que obtengamos sea 1. Todos los valores que se obtienen, son los exponentes intermedios de X que se van obteniendo en la segunda localidad de memoria. Escrito de otro modo, usamos el hecho de que:

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

Imagen 1 [3]

A continuación, se presenta nuestra implementación del algoritmo en un script muy sencillo escrito en Python. Nótese que en la función, la entrada `x` denota el exponente y la entrada `n` denota el número de multiplicaciones realizadas. No confunda el exponente `x` de la función de python con las base X a la que nos hemos estado refiriendo previamente.

```
# Función que imprime la secuencia de exponentes por los que se debe pasar
para llegar al resultado deseado de manera óptima.
# x.- el exponente que se desea analizar
# n.- el número de multiplicaciones hechas hasta el momento. Debe iniciar en
0
# Salida.- El número de multiplicaciones que se deben hacer para llegar al
resultado
def sequence(x,n):
      if x is not 1:
            if x%2 == 0:
                  n = sequence(x/2,n)
            else:
                  n = sequence(x-1,n)
            print x
```

```
        return n+1
    else:
        return 0
```

Posteriormente a nuestra inferencia, nos dimos cuenta de que el algoritmo que se nos ocurrió para resolver el problema tiene el nombre de exponenciación (potenciación en realidad) binaria o “Exponentiation by squaring”. Dicho algoritmo nos arroja la secuencia de exponentes que se deben obtener para llegar al resultado deseado con el menor número de multiplicaciones utilizando sólo dos localidades de memoria y es vastamente utilizado.

*2) Usando **m** localidades de memoria:*

Nuestro enfoque inicial para atacar este problema usaba la factorización prima de los exponentes dados. De ese modo, primero se buscaba formar uno de los componentes primos del exponente para luego elevar al cuadrado tantas veces fuera necesario y llegar al resultado deseado. Sin embargo, esto no siempre nos daba una mejor solución que el algoritmo de exponenciación binaria. Por ejemplo, para el caso de 631 que es un número primo. Aplicamos la descomposición en factores primos a 630 y el resultado final lo multiplicamos por X.

Analizando más detalladamente el problema menos restringido en memoria, nos dimos cuenta de que en realidad, este problema se reduce a encontrar la suma encadenada más corta que genera el exponente n. Éste es un problema NP-completo. Existen varios algoritmos y heurísticas que dan un resultado suficientemente bueno.

Uno de ellos, bastante interesante, es el algorìtmo de Pippenger (ver [2]) el cual busca realizar la potenciación utilizando un número pequeño de multiplicaciones. Dicho algoritmo utiliza vectores en el intervalo [0,1] y obtiene el resultado deseado realizando sumas de vectores comenzando por el vector unitario. Este algoritmo recibe dos parámetros el primero el conjunto de exponentes que se quieren descomponer y el segundo un grupo de grupo de vectores en los que se almacenan la descomposición de cada exponente. El resultado se da como $X(S_i)$ y en cada i-ésima posición está la descomposición de un exponente de la entrada.

El algoritmo de Pippenger en general se puede ver como un algoritmo recursivo parametrizado que llega a un nivel dado de recursión una serie de valores de entrada y en algunos casos un pivote. Comienza por dividir la entrada por un factor c y posteriormente agrupa el resultado con parámetros a_1 y b_2, posteriormente agrupa la entrada con parámetros a_1 y b_2 . Este proceso se realiza hasta n niveles de recursión de tal suerte que cada producto pueda realizarse de manera individual.

II. MATRICES

Dadas las matrices $A_1 \times A_2 \times A_3 \times A_4 \times \ldots\ldots\ldots \times A_n$ donde las matrices tienen dimensiones

compatibles-pero no son cuadradas- encontrar la forma de minimizar el tiempo de computación.

Este problema es de optimización y puede ser resuelto utilizando programación dinámica. Dada una secuencia de matrices, queremos encontrar la manera más eficiente de realizar las multiplicaciones entre ellas. El problema no es realizar las multiplicaciones sino en decidir el orden en que se deben efectuar dichas operaciones.

Debido a que la operación de la multiplicación de matrices es asociativa, se tienen diversas opciones para realizar las multiplicaciones. Sin embargo el orden con que se asocian los productos afecta el número de operaciones aritméticas que se necesitan para computar el producto o eficiencia.

Ejemplo:

Si se tuvieran tres matrices en las que A es una matriz de 10 x 30, B es una matriz de 30 x 5 y C es una matriz de 5 x 60 entonces:

de operaciones requeridas de (AB)C = (10 x 30 x 5) + (10 x 5 x 60) = 4,500 operaciones
de operaciones requeridas de A(BC) = (30 x 5 x 60) + (10 x 30 x 60) = 27,000 operaciones

Del ejemplo anterior, podemos observar que la primera manera de realizar la multiplicación de las matrices resulta mucho más óptima que la segunda ya que para la segunda se realizan 6 veces más operaciones.

A. Más ejemplos

Sean:
A1 = [100,4]
A2 = [4,50]
A3 = [50,20]
A4 = [20,100]

Matrices de dimensión m x n. A partir de ellas debemos de encontrar la forma óptima de multiplicarlas para producir el menor número de operaciones.

Sabemos que hay diversas formas de multiplicar las matrices A1, A2, A3 y A4 si se considera la forma de agrupar las matrices, el orden en el que se agrupan y tomando en cuenta que la forma de multiplicar las matrices debe respetar la compatibilidad de las matrices. De las posibles soluciones una resultará la más óptima y otra la menos óptima. Nosotros únicamente mostraremos la solución más y menos óptima. El factor para comparar la eficiencia fue el número de operaciones realizadas.

1) Solución óptima:

De realizar el análisis de todas las posibles soluciones, se llegó a que la forma óptima de realizar la multiplicación de las matrices es la siguiente:

A1((A2A3)A4)
El número de operaciones requeridas al realizar esta multiplicación es la siguiente:

A1((A2A3)A4) = (4 * 50 * 20) + (4 * 20 * 100) + (100 * 4 * 100) = 52,000

2) Solución menos óptima:

Similarmente, se encontró que la manera menos eficiente de multiplicar las matrices anteriores es la siguiente:

((A1A2)(A3A4))

Para este caso, se realizarían 620,000 operaciones que se pueden obtener de desarrollar la siguiente expresión:

((A1A2)(A3A4)) = (100 * 4 * 50) + (50 * 20 * 100) + (100 * 50 * 100) = 620,000

B. Nuestra propuesta de algoritmo

Por lo que hemos podido ver al resolver este problema, podemos identificar rápidamente un punto que nos parece fundamental. Al hacer el producto de dos matrices, la primera de dimensiones m x n y la segunda de dimensiones n x r debemos realizar m x n x r operaciones y la matriz resultante es de m x r. ¿Qué es lo que sucede? Desaparece la componente de "en medio" de las matrices y permanecen las dos componentes exteriores para volver a ser tomadas en cuenta en el siguiente producto de matrices.

Por esta razón, vemos que es más óptimo multiplicar las matrices de modo tal que se realicen primero las multiplicaciones que tienen como componentes centrales los números más grandes. Así, éstos sólo estarán tomados en cuenta una vez en el producto de matrices.

Regresando al ejemplo de la sección anterior, podemos ver que en el caso más óptimo, se multiplican primero las matrices que hacen que el valor más grande de dimensión (en este caso 50) quede atrapado en medio. Nótese que las dimensiones con valor de 100 deben forzosamente quedar en los extremos. Por eso no las consideramos. La siguiente multiplicación que se realiza tiene dimensión intermedia de 20.

En el caso menos óptimo se hace justo lo inverso. La primera multiplicación "desaparece" el valor de 4 y deja en los extremos valores de 100 y 50 sólo para que vuelvan a ser tomados en cuenta en la siguiente multiplicación.

Así, nuestra primera aproximación a un algoritmo para resolver este problema pretende multiplicar primero matrices que tengan la dimensión intermedia más grande y las dimensiones más pequeñas en los extremos. Sin embargo, entendemos que para cadenas más grandes de matrices, es posible que este algoritmo no nos dé necesariamente la solución óptima. Con vistas a poder hacer determinado producto de matrices más adelante, es posible que sea preferible dejar ciertas dimensiones grandes en los extremos en algún producto inicial.

De cualquier modo, creemos que es una buena idea. Quizás se podría utilizar como "caso base" para operar en secuencias pequeñas generadas a partir de una subdivisión del problema original aplicando la filosofía de "divide y vencerás".

C. Programación dinámica

En las ciencias de la computación, programación dinámica se refiere a un método que se utiliza para resolver problemas complejos descomponiéndolos en partes más pequeñas. En general la estrategia a seguir es descomponer un problema en subproblemas, luego ir resolviendo esos subproblemas para luego combinar sus respectivas soluciones y así llegar a la solución final. Una forma de implementar una estrategia de este tipo es haciendo un programa recursivo, el cuál va haciendo llamadas así mismo con parámetros que hacen que el problema se divida en partes más pequeñas, hasta que el problema es tan sencillo que el programa puede dar la respuesta a ese problema sencillo y partiendo de esta solución, el programa va regresando en las llamadas recursivas combinando las respuestas que ya se tienen. Por lo tanto, un programa recursivo consta de dos partes principales, una que es un caso base que consiste en que el programa ha descompuesto el problema hasta un grado tan sencillo que el programa ya sabe la respuesta, la segunda parte consiste en las llamadas recursivas al programa para descomponer el problema en subproblemas.

Regresando al tema y al problema que se quiere resolver hay que dejar claro que se quiere encontrar es la secuencia de multiplicaciones que tiene el costo mínimo o el número mínimo de operaciones aritméticas que se realizan.

El caso base para establecer la recursividad será la multiplicación de dos matrices en el que sólo hay una manera de realizar la operación y por lo tanto el costo mínimo es el costo de realizar esta operación. En general, se puede encontrar el costo mínimo utilizando el siguiente algoritmo recursivo:

1. Se toma la secuencia de matrices y se separa en dos subsecuencias de matrices.
2. Encontrar el costo mínimo de multiplicar esas subsecuencias por separado.
3. Sumar los dos costos de las subsecuencias y además agregar el costo de multiplicar las subsecuencias entre sí.
4. Realizar esto para cada posición en la cuál la secuencia de matrices original puede ser separada y al final tomar el costo mínimo de entre todas las opciones.

Por ejemplo si tenemos cuatro matrices *ABCD* , computamos el costo de (A)(BCD), (AB)(CD), (ABC)D haciendo llamadas recursivas para encontrar el mínimo costo para computar ABCD. Escogemos la mejor solución. Sin embargo esto puede llegar a ser muy ineficiente porque se recorren todas las permutaciones y se realiza trabajo redundante. Para resolver este problema recurrimos a la técnica de "memorización".

La "memorización" consiste en que cada vez que se computa el costo mínimo de una subsecuencia específica, éste se guarda. Si en algún momento se vuelve a preguntar por ese costo mínimo simplemente se da la respuesta ya guardada y no la computamos. Considerando que hay $n^2/2$ subsecuencias, donde "n" es el número de matrices en la secuencia, entonces el espacio requerido para guardar los valores de los costos es razonable. Este truco hace que el tiempo de ejecución del algoritmo cambie del $O(2^n)$ a $O(n^3)$.

Esto último sucede porque sin el truco tendríamos que averiguar todas las posibilidades, es decir, todas las permutaciones y estaríamos haciendo mucho trabajo redundante. Por ejemplo, si queremos hacer una llamada recursiva para encontrar el mínimo costo para computar *ABC*(matrices) y *AB*(también matrices),

forzosamente debemos encontrar el costo de *AB* para encontrar el costo de *ABC*. Sin la memorización, se estaría haciendo doble trabajo primero para encontrar el valor de *AB* y luego para encontrar el valor de *ABC*, con la memorización, al conocer el valor del costo mínimo para *AB*, se guarda en memoria y cuando llega el momento de calcular *ABC*, parte del problema ya está resuelto porque ya sabemos el resultado para *AB* y ya sólo se tendría que calcular el costo mínimo de la parte restante del problema y por lo tanto el tiempo de ejecución es menor.

REFERENCIAS

[1] Cormen, Thomas H, et. al. “Introduction to Algorithms”. Second Edition. MIT Press and McGraw-Hill. pp. 331–338. 2001.

[2] Daniel J. Bernstein, "Pippenger's Algorithm”, http://cr.yp.to/papers/pippenger.pdf, consultado el 19/09/2013

[3] Wikipedia contributors, 'Exponentiation by squaring', Wikipedia, The Free Encyclopedia, 13 September 2013, 17:02 UTC, <http://en.wikipedia.org/w/index.php?title=Exponentiation_by_squaring&oldid=572780260> [accessed 20 September 2013]