Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2283 - Diseño y Análisis de Algoritmos
Profesor: Nicolás Van Sint Jan
Ayudante: Dante Pinto

Ayudantía 4

Programación dinámica

Problema 1

Königsberg era una ciudad en Prusia, hogar de múltiples matemáticos famosos, y hogar del problema de los puentes de Königsberg, un problema clásico de teoría de grafos el cual consiste en encontrar un camino en Königsberg que pase por cada uno de sus siete puentes exactamente una vez.

Euler, en 1736, dio la solución a este problema de forma negativa, no existe algún camino que cumpla esas condiciones. Hay más preguntas que se pueden hacer sobre el multigrafo con el cual se abstrae el problema, mostrado en la Figura 2. Una de ellas es contar la cantidad de caminos que pasan por N aristas con la posibilidad de repetir aristas. Este último es el problema que tienen que solucionar, específicamente sobre el multigrafo de la Figura 2.

En otras palabras, deben diseñar un algoritmo que, dado un número N, debe retornar la cantidad de caminos de largo N existentes en el multigrafo, considerando que se pueden repetir aristas en los caminos. La complejidad del algoritmo dado debe estar en $\mathcal{O}(\log(N))$

Figure 1: Puentes de Königsberg

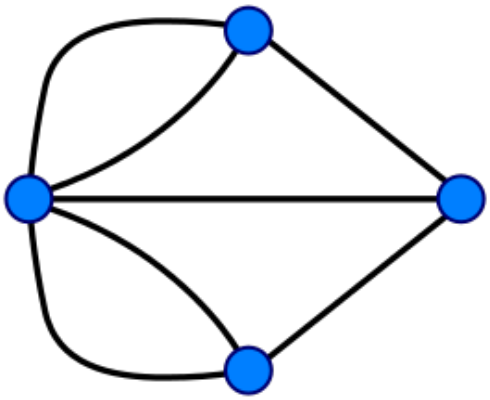

Figure 2: Multigrafo asociado al problema

Solución: Es claro que la cantidad de caminos en el grafo será igual a sumar la cantidad de caminos salientes desde cada nodo. Observando el caso base para $N = 1$, podemos ver que la cantidad de nodos que tendremos será:

$$n^N = 5 + 3 + 3 + 3 = 14$$

Lo que es consistente con la cantidad de caminos que podemos contar.

Observando ahora los caminos de largo 2, podemos ver que la cantidad de caminos salientes de cada nodo serán la cantidad de caminos salientes de tamaño 1 por la cantidad de caminos salientes de cada uno de sus vecinos, ponderados por la cantidad de aristas desde el nodo a dicho vecino.

Siguiendo esta misma idea, los caminos de largo N salientes de un nodo serán los caminos de largo $N-1$ de sus vecinos por la cantidad de aristas de cada vecino. Asignando un orden arbitrario a los nodos (de izquierda a derecha y arriba a abajo en la figura 2) tenemos:

$$n_1^N = 2 \cdot n_2^{N-1} + 2 \cdot n_3^{N-1} + n_4^{N-1}$$
$$n_2^N = 2 \cdot n_1^{N-1} + n_4^{N-1}$$
$$n_3^N = 2 \cdot n_1^{N-1} + n_4^{N-1}$$
$$n_4^N = n_1^{N-1} + n_2^{N-1} + n_3^{N-1}$$

$$\Rightarrow n^N = n_1^N + n_2^N + n_3^N + n_4^N$$

Con las recursiones anteriores podemos ver que solamente necesitamos guardar un resultado anterior en memoria, pues cada n_i^N depende solamente de los valores para $N-1$, sin embargo se puede ver también que para obtener el N-ésimo resultado, simplemente tendremos que aplicar cada una de las 4 recursiones $N-1$ veces a los valores de base.

Esta estrategia se traduce perfectamente a una matriz, M, en que las filas representan los valores de n_i^N y las entradas son los pesos de las recursiones anteriores, de esta manera, para obtener n^N bastará multiplicar el vector de pesos iniciales, v, por M^{N-1} y luego sumar sus entradas. Dados los valores del ejercicio, M y v_1 son:

$$M = \begin{bmatrix} 0 & 2 & 2 & 1 \\ 2 & 0 & 0 & 4 \\ 2 & 0 & 0 & 4 \\ 1 & 1 & 1 & 0 \end{bmatrix} \qquad v = \begin{bmatrix} 5 \\ 3 \\ 3 \\ 1 \end{bmatrix}$$

$$\Rightarrow v_N = M^{N-1} \cdot v_1$$

Finalmente, para lograr la complejidad logarítmica que buscamos, bastará con usar el algoritmo de exponenciación rápida para calcular M^{N-1}, que tendrá una complejidad logarítmica respecto a N, pues, dado que tanto la multiplicación de M por v como la suma de entradas del vector resultante tendrán una cantidad constante de operaciones, el tiempo del algoritmo estará dado por el tiempo de cómputo de M^{N-1}, es decir, $\mathcal{O}(\log(N))$.

Problema 2

Dadas dos matrices de números enteros, $A_{f\times g}$ y $B_{g\times h}$, el algoritmo usual para calcular $A \cdot B$ realiza $f \cdot g \cdot h$ multiplicaciones de números enteros.

Diseñe un algoritmo que dada una secuencia de m matrices, A_1, A_2, ..., A_m entregue el número mínimo de multiplicaciones necesarias para encontrar $A = A_1 \cdot A_2 \cdot ... \cdot A_m$ en tiempo polinomial.

Solución: Sean $f(A)$ y $c(A)$ funciones que entregan la cantidad de filas y columnas de la matriz A, respectivamente. Definimos la recursión:

$$\texttt{NMul}(A_1, ..., A_m) = \begin{cases} 0 & m = 1 \\ \min\limits_{1 \le k < m} \big(\texttt{NMul}(A_1, ..., A_k) & \\ \qquad +\texttt{NMul}(A_k, ..., A_m) & \\ \qquad +f(A_1) \cdot c(A_k) \cdot c(A_m) \big) & m > 1 \end{cases}$$

Esta recursión prueba exhaustivamente todas las posibles combinaciones de pares de matrices para la multiplicación, pues itera el valor de i probando todas las posibles divisiones, resolviendo cada una de forma recursiva y quedándose con el valor mínimo entre todas ellas.

A pesar de que el algoritmo anterior funciona, es claro que no se ejecuta en tiempo polinomial, pues cada llamada recursiva itera sobre todos los posibles i y además hace más llamadas en cada una de estas iteraciones.

A pesar de lo anterior, es fácil ver que tenemos una enorme cantidad de llamadas repetidas, por lo que podemos optimizar este algoritmo usando programación dinámica. Para esto, necesitaremos una tabla M de $m \times m$, donde iremos almacenando los resultados de `NMul`, de forma que $M[i, j] = \texttt{NMul}(A_i, ..., A_j)$.

Usando esta tabla, podemos utilizar el siguiente algoritmo para llenarla iterativamente y obtener el resultado:

1. Llenamos la diagonal de ceros.
2. Llenamos las entradas de $M[i, i+1]$ para las que tendremos que realizar solamente dos multiplicaciones en cada una, dadas por el último término del paso recursivo $f(A_1) \cdot c(A_i) \cdot c(A_m)$.
3. Llenamos las entradas de $M[i, i+2]$ para las que tendremos que realizar cuatro multiplicaciones, correspondientes a calcular $\texttt{NMul}(A_i, A_{i+1}) + f(A_i) \cdot c(A_{i+1}) \cdot c(A_{i+2})$ y $\texttt{NMul}(A_{i+1}, A_{i+2}) + f(A_i) \cdot c(A_{i+1}) \cdot c(A_{i+2})$.
4. Repetimos el proceso $k = m - i - 2$ veces más, llenando las entradas de $M[i, i+k]$ y realizando $2k$ multiplicaciones en cada celda.
5. Retornamos $M[1, m]$, que corresponde a $\texttt{NMul}(A_1, ..., A_m)$

En el peor de los casos, haremos $2m$ multiplicaciones por celda (lo que solo ocurre en la celda $M[1, m]$) y, dado que tenemos m^2 celdas, la complejidad del algoritmo estará en $\mathcal{O}(m^3)$, por lo que tenemos un algoritmo polinomial.

Cabe señalar que para esta complejidad estamos considerando como operación básica la multiplicación de números enteros, pero incluso si agregamos las operaciones de suma de enteros y lecturas/escrituras en la tabla, seguiremos teniendo una polinomial de operaciones, pues al igual que para las multiplicaciones, la cantidad de sumas y la cantidad de accesos a la matriz será de orden lineal en cada celda y escribiremos cada celda una sola vez.