Problema 3: Tree Nesting

Rodrigo García Gómez RODRIGO.GARCIA@ESTUDIANTES.MATCOM.UH.CU

Tutor(es):
Alfredo Somoza

Resumen

Dados dos árboles S y T se debe encontrar el número de subárboles de S que son isomorfos con T
Como es costumbre, primeramente se enfocará el problema a la manera de la fuerza bruta de fácil demostración, para tener con qué comparar las respuestas de cualquier solución siguiente.
El algoritmo de fuerza bruta calcula todos los subtrees de S y luego verifica cuántos de estos cumplen con la propiedad de ser isomorfos con T.
El algoritmo optimizado toma cada nodo de T y lo hace raíz del árbol, luego, para cada uno de estos árboles enraizados se crea una máscara de bits. En cada nodo v habrá una lista que tendrá 1 en la posición k, si el nodo k está directamente conectado con v, y 0 si no lo está. Teniendo esta información almacenada se realiza un algoritmo recursivo sobre S para calcular la respuesta, en el cuál se usará programación dinámica. De esta forma se almacenan valores previamente calculados y se evita repetir cálculos innecesarios.

1. Texto del problema

You are given two trees (connected undirected acyclic graphs) S and T.

Count the number of subtrees (connected subgraphs) of S that are isomorphic to tree T. Since this number can get quite large, output in modulo 109+7.

Two subtrees of tree S are considered different, if there exists a vertex in S that belongs to exactly one of them.

Tree G is called isomorphic to tree H if there exists a bijection f from the set of vertices of G to the set of vertices of H that has the following property: if there is an edge between vertices A and B in tree G, then there must be an edge between vertices $f(A)$ and $f(B)$ in tree H. And vice versa — if there is an edge between vertices A and B in tree H, there must be an edge between $f^1(A)$ and $f^1(B)$ in tree G.

2. El problema

Dados dos árboles (trees) S y T, se debe contar el número de subárboles (subtrees) de S que son isomorfos con T. La respuesta debe ser dada en módulo de $10^9 + 7$. Dos subtrees de S se consideran diferentes si existe algún vértice de S que pertenece a exactamente uno de ellos.

Dos trees S y T son isomorfos si existe alguna función f biyectiva, que vaya de del conjunto de los vértices de S al de los vértices de T y que cumpla con lo siguiente:

- Para cualquier arista $< u, v >$ que pertenece a S, entonces la arista $< f(u), f(v) >$ debe pertenecer a T.
- Para cualquier arista $< f(u), f(v) >$ que pertenece a T, entonces la arista $< u, v¿$ debe pertenecer a S.

Se recibe como entrada un entero n, que representa la cantidad de vértices de S, seguido de una lista de $n - 1$ parejas de enteros que

representan sus aristas y un entero m que representa la cantidad de vértices de T, seguido de una lista de $m-1$ parejas de enteros que, nuevamente, representan sus aristas. El valor de n está restringido entre 1 y 1000 y el de m entre 1 y 12.

2.1 Algunas observaciones

La definición de subtree utilizada será la de teoría de grafos, un subtree de A, es un subgrafo del grafo (árbol) A, que es también un árbol.

A partir de este punto se utilizará mucho el término enraizar el árbol. Un árbol A, enraizado en un vértice v es el mismo árbol, manteniendo todos los vértices y aristas, pero viendo a v como la raíz. Esto significa que v tendrá como hijos a todos sus vértices adyacentes y sus adyacentes tendrán como hijos a sus ropios adyacentes menos a A (esto es recursivo para todos los nodos). Por simplicidad y comodidad se llamará también rooted tree.

En ambos algoritmos se utiliza una clase creada de nombre `Tree`, esta será una manera útil de almacenar la información de los árboles y contará con algunos métodos y algunas propiedades explicadas a continuación:

- La propiedad `sons` almacenará en la posición k, los hijos del nodo k. los valores guardados en sons dependerán de quién sea la raíz en cada momento.
- Se guarda `n`, la cantidad de vértices del árbol.
- La propiedad `edges` tendrá en la posición u, v el valor 1 si existe arista entre u y v, y 0 en caso contrario.
- La propiedad `bitmask` tendrá la máscara de bits necesaria para la solución óptima. Sus valores, al igual que en `sons` dependerán de la raíz actual.
- La función `make_roots` se utiliza para enraizar el árbol, poniendo como raíz a la entrada `root`.

Es necesario mencionar al usuario de Codeforces con alias "*star_cried*" cuya explicación fue imprescindible para iluminar un problema que llevaba 5 días en tinieblas.

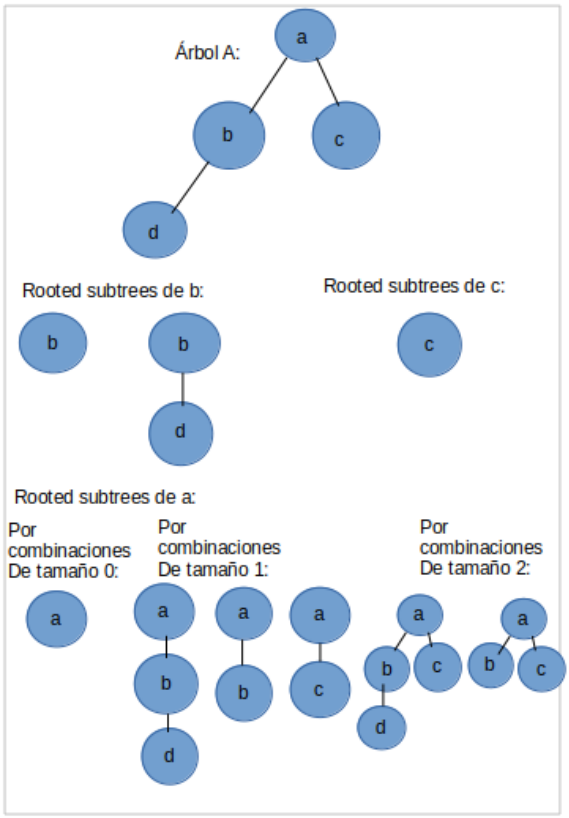

Figura 1: Ejemplo de obtención de rooted subtrees de un árbol

3. Idea Fuerza Bruta

Los subtrees cuya raíz es v se pueden encontrar calculando recursivamente, para cada uno de los hijos u de v, todos los subtrees que tienen a u como raíz. Una vez estos subtrees calculados, se toma a todas las combinaciones posibles de hijos de v (combinaciones de tamaño k, para $0 <= k <=$ `len(sons[v])` y, teniendo una combinación, para cada forma de escoger a un rooted subtree de cada hijo de esta combinación, se forma un subtree con raíz en v poniendo a los rooted subtrees como hijos de v. En la Figura 1 se puede ver un ejemplo de este proceso.

Una vez obtenidos todos los subtrees, se crea una lista `mask`, de tamaño `len(subtrees)` que tendrá, en la posición k, el valor 1 si el subtree en la posición k de subtrees es isomorfo con s, y 0 en caso contrario. Para dar la respuesta basta con contar la cantidad de posiciones en

`mask` que tienen valor 1.

Para llenar $mask$, es necesario enraizar a T en cada uno de sus vértices, y cada vez que se haga esto, se verificará para cada subtree si T es isomorfo enraizado (en la Definición 1 se explica cuando dos árboles son isomorfos enraizados) con él y se pondrá a la posición de ese subtree en mask el valor 1 . Note que para los subtrees con valor 1 en mask no es necesario hacer el cálculo nuevamente.

Para verificar si dos rooted trees a, b son o no isomorfos enraizados se utiliza un método recursivo que se basa en lo siguiente.

Definición 1: a y b rooted trees son isomorfos enraizados si y sólo si:

1. a y b no tienen hijos (ambos).
2. a y b tienen la misma cantidad de hijos (k) y sean $a_hijos = a_1, a_2, \ldots, a_k$ y $b_hijos = b_1, b_2, \ldots, b_k$ los hijos de a y b respectivamente. Existe una permutación p de b_hijos tal que el hijo p_i de esa permutación es isomorfo enraizado con el hijo ai de a_hijos, para todo $i1 <= i <= k$.

Teorema 1: A y B dos árboles son isomorfos si y sólo si el rooted tree (a) que se forma al enraizar a A en un punto arbitrario, es isomorfo enraizado con al menos uno de los rooted trees (b) que se forman al enraizar a B con cualquiera de sus vértices.

$\Leftarrow$ Demostrando que si a es isomorfo enraizado con algún b, entonces A y B son isomorfos. Como a y b son isomorfos enraizados, entonces se cumple 1 o 2. Como se cumple 1 (a y b tienene ambos 0 hijos), a y b son isomorfos, pues $f(r_a) = r_b$, siendo r_a y r_b las raíces de a y b respectivamente. Como son los únicos nodos de a y b, y en ninuno existe aristas, entonces las condiciones de f para el automorfismo se cumplen.

Si se cumple 2, entonces nuevamente $f(r_a) = r_b$, pero además $f(p_i) = a_i$. Como r_b está unido con todos los p_i y ra está unido con todos los a_i, entonces estas aristas cumplen las condiciones de f para isomorfismo. Luego se tiene también que todos los b_i son isomorfos con los a_i y por tanto basta con unir las funciones que cumplen con estos isomorfismos a f.

Es trivial ver que si a y b son isomorfos, entonces A y B también lo son pues a comparte todas sus aristas y nodos con A y b lo hace con B.

$\Rightarrow$ Demostrando que si A y B son isomorfos, entonces existe algún b tal que a y b son isomorfos enraizados.
Sea f una función que cumple con el isomorfismo de A y B. Se puede enraizar a A en un punto arbitrario v, y a B en $f(v)$, los rooted trees que se forman al hacer esto, serán a y b. Ahora, es necesario demostrar que a y b cumplen con 1 o con 2 y que por tanto son isomorfos enraizados. Se demostrará que en caso de no cumplirse 1, entonces necesariamente se cumplirá 2.

Que no se cumpla 1, implica que a y b tienen hijos. Si no tuvieran la misma cantidad de hijos, entonces f no pudiera cumplir con la condición de isomorfismo pues existiría un u tal que existiera la arista $< v, u >$ en A y no la $< f(v), y >$ en B o viceversa. Luego sean $a_hijos = a_1, a_2, \ldots, a_k$ y $b_hijos = b_1, b_2, \ldots, b_k$ los hijos de v y $f(v)$ respectivamente, existe una permutación p de b_hijos tal que el hijo pi de esa permutación es isomorfo enraizado con el hijo ai de a_hijos, para todo $i1 <= i <= k$. Esta permutación será: $f(a_1), f(a_2), \ldots, f(a_3)$.

La correctitud del algoritmo se basa en el Teorema 1 ya que cada subtree de subtrees será isomorfo con T si y sólo si, es isomorfo enraizado con alguno de los rooted trees que se forman al enraizar a S en cada uno de sus nodos. Además, no existe el peligro de dejar sin comprobar un subtree de S isomorfo con T, pues todos son verificados.

Es evidente que este algoritmo es extrema-

damente ineficiente. La cantidad de subtrees de un árbol es exponencial con respecto a la cantidad de nodos y luego, por cada uno de esos subtrees calculados, se verifica el isomorfismo con una función que se llama m veces y hace un ciclo que verifica en el peor caso $(m-1)!$ Permutaciones. Sin embargo, como se ha mencionado antes, será útil para el proceso de testeo tener los resultados devueltos por un algoritmo de relativamente fácil demostración, para así poner a prueba el algoritmo con una complejidad óptima.

4. Algoritmo óptimo

Primeramente se enraíza a S en un punto arbitrario (se escogió 0) para tener calculados a los hijos de cada nodo. Como ya se mencionó anteriormente, se utilizará una máscara de bits en T, cada nodo v tendrá una lista de tamaño m, que estará marcada con 1 en la posición k si el vértice k es un hijo de v y 0 en caso contrario. La lista que almacena estos bits se creará cada vez que se enraíza al árbol T. La solución (que se almacena en `ans1`) se calculará sumando los resultados de la función calculate ejecutada sobre T una vez por cada forma de enraizarlo, o sea, enraizado en cada uno de sus vértices. Se analizará la función, sin perder generalidad, para T enraizado en un nodo v.

El método calculate se ejecutará sobre cada nodo k de S, y su función es analizar cuántos subtrees de S enraizados en k son isomorfos enraizados con T, de raíz v. Esto se puede interpretar también como analizar la cantidad de maneras en que el árbol T encaja en el árbol S, desde k hacia abajo, matchando a k con la raíz v de T. La lista de bits inicialmente será la alacenada en T para el nodo v, y la variable son tendrá el número en que está almacenado el hijo u de k (o sea, la posición de la lista `sons[k]` de el árbol S).

El proceso recursivo que sigue el algoritmo es el siguiente: Se matchea a v con k y luego se verifica de cuantas formas puede matchear cada hijo u de k con los hijos de v (que están representados por `bits`). Para cada hijo u, se calcula el resultado de no tenerlo a él en cuenta para matchear y se le adiciona a esto, el resultado de matchearlo con cada uno de los lugares que queden con valor 1 de la lista de bits. El resultado de matchear a u con un bit en la posición b de `bits` será igual al resultado de aplicarle este proceso a los hermanos de u, dándole a `bits` con valor 0 en su posición b (apagando el bit b), multiplicado por el resultado de aplicarle el proceso a cada uno de los hijos de u, pasándole como bits a la lista de bits almacenada en la posición b de los bitmasks de T. Cuando han sido analizados todos los hijos de un nodo, entonces se alcanza el caso base y, si todos los bits fueron matcheados (No hay ninguno con valor 1), entonces se toma como un matceho positivo y se retorna 1, en caso contrario es un matcheo negativo y se retorna 0.

Demostrando que al aplicar este proceso, todos los subtrees enraizados en k, isomorfos con T enraizado en v, son contados.

Sea s un subtree que cumple esta condición, s debe tener una de las siguientes formas:

1. Es un árbol hoja.
2. k tiene una cantidad c de hijos, todos hojas.
3. k tiene una cantidad c de hijos, con al menos uno no hoja.

*Caso*1:
Será contado cuando se analice el caso base. Todos los hijos de k, ya que no tiene ninguno, y retornará 1, pues como v tampoco tendrá hijos, su máscara de bits no tendrá ningún valor 1.

*Caso*2:
En este caso, k tiene tantos hijos como bits con valor 1 hay en la máscara de v. Por tanto el primer hijo analizado matchea con un 1, el segundo con otro, ..., y, cuando se llega al caso base, todos los bits fueron matcheados, por tanto se cuenta.

*Caso*3:
Como s y T son isomorfos enraizados no hojas, k y v tienen la misma cantidad de hijos y existe una permutación $p_1, p_2, \ldots, p_c$ de los hijos de v $(v_1, v_2, \ldots, v_c)$ tal que p_i es isomorfo enraizado con v_i para todo i, $1 <= i <= c$. Como el algoritmo analiza todas las formas en que los hijos de k matchean con los bits encendidos, entonces se analizará el caso en que v_i matchea con p_i. Los hijos no hojas serán analizados recursivamente siguiendo este mismo proceso, y serán de alguno de estos tres casos y, como ellos son también isomorfos enraizados entre ellos, entonces retornarán al menos un matcheo positivo. Cuando se llegue al caso base, todos los bits habrán sido matcheados y se retornará un matcheo positivo.

La función `calculate` realiza este proceso explicado. El primer llamado a `calculate` sirve para calcular las soluciones sin tener en cuenta al hijo u que está siendo analizado actualmente y el ciclo for que analiza los hijos de T analiza los matcheos de u con cada uno de estos hijos y multiplica el enálisis de los hermanos de u, una vez este matcheado con un hijo con los matcheos de los hijos de u. Además, se utilizan las listas `visited` y `dp` para almacenar los valores ya calculados y evitar recalculos innecesarios. Al entrar a un llamado recursivo, si el hijo actual bajo análisis, con la máscara de bits que se tiene actualmente han sido ya visitados, entonces se busca su valor en `dp` y se retorna este.

Es fácil notar que varios de los subtrees de S están siendo contados más de una vez. Esto se debe a lo siguiente: Suponga que el subtree s matchea con T enraizado en v, v_1 y v_2 son hijos de v y son isomorfos entre sí. Para matchear s y T enraizado en v, v_1 y v_2 tienen que haber matcheado con dos hijos de la raíz de s a los que se llamará s_1 y s_2 respectivamente. Como v_1 y v_2 son isomorfos, entonces también se podrá matchear a v_1 con s_2 y a v_2 con s_1. Además, si al enraizar a T por un nodo v, el rooted tree resultante fuera isomorfo enraizado con el resultante de enraizar a T por un nodo u, entonces cada uno de los matcheos positivos retornados por el análisis de T enraizado en v, serán también retornados por el análisis de T enraizado en u.

El culpable del problema de las cuentas múltiples es el número de automorfismos de T.

Definición: Un automorfismo enraizado T' de T enraizado en v es una forma de enraizar a T en cualquiera de sus vértices y de ordenar a los hijos de cada nodo de cualquier forma, de manera que se cumplan las condiciones de isomorfismo enraizado.

Para demostrar que si un subtree s de S matchea con T enraizado en v, entonces matcheará también con cualquier automorfismo enraizado T' de T enraizado en v, basta con notar que si A es isomorfo enraizado con B y B lo es con C, entonces A lo será con C. O sea, la definición de isomorfismo enraizado transitiva. Como por definición T' es isomorfo enraizado con T, entonces también lo será con s pues el algoritmo matcheó a s con T por ser estos isomorfos enraizados. Luego, como ya se demostró que el algoritmo matchea a todos los árboles isomorfos enraizados con cada subtree de S, entonces T' también se matchea.

Esto demuestra que por cada uno de los automorfismos enraizados que existan en T con raíz en v, todos los subtrees de S isomorfos enraizados con T son contados una vez más. Para hallar el número de automorfismos enraizados de T basta con realizar el mismo proceso anterior, pero esta vez matcheando los isomorfismos de T sobre el mismo árbol T (se hace $S = T$). Sólo es necesario calcular los automorfismos enraizados de T con raíz en un vértice arbitrario, pues la cantidad será igual al resultado de enraizarlo en cualquier otro. Esta vez no es necesario analizar los subtrees enraizados en vértices diferentes a la raíz de S, pues S y T tienen la misma cantidad de nodos, si se deja de tener en cuenta a alguno ya el isomorfismo no sería posible.

Una vez se tiene el número de automorfismos enraizados de T, basta con con dividir la

respuesta antes calculada (`ans1`) por la cantidad de automorfismos (`ans2`) para obtener la solución final. Note que esta solución no saldrá de los enteros, ya que por cada automorfismo enraizado se cuenta la misma cantidad de respuestas (todas las existentes), y por tanto estos números serán múltiplos. Además la cantidad de automorfismos enraizados no será 0, pues el propio T con raíz en v siempre será uno.

Análisis de complejidad temporal: A lo sumo habrá una cantidad 2^m de máscaras de bits distintas, y gracias a la programación dinámica empleada, la recursión de calculate sólo hará que esta función se ejecute un máximo de $n*2^m$ veces, que es la cantidad de parejas de vértices con máscaras de bits distintas que pueden existir. Dentro de *calculate* se ejecuta un ciclo de tamaño m (caso base) y uno de tamaño n, no anidados. Y la función se llama desde el método solve, dentro de dos ciclos anidados de n y m iteraciones. Todo esto indicaría que la complejidad temporal del algoritmo pertenece a $O(n * m * n * 2^m * (n + m))$, sin embargo, la entrada del valor m en el problema está limitada a un máximo de 12. A pesar de que 2^12, que es el peor caso de m, es un valor considerablemente grande, sigue siendo constante y por tanto se considerará a m como un valor constante. Luego, la complejidad temporal del algoritmo termina perteneciendo a $O(n^3)$.