Curso de Programación en C++.
TPL2-2021. Trabajo Práctico de Laboratorio. [2021-06-23]

Ejercicios

Supongamos que tenemos un conjunto de vectores **`vector<vector<int> > vectors`** y queremos buscar cuantos de ellos son diferentes entre si. Por ejemplo
`vectors=[[1,0],[1,1],[0,0],[1,0],[1,1]] => 3 diferentes`
`vectors=[[3,4],[1,2],[1,2],[3,4]] =>  2 diferentes`
Entonces la solución "por fuerza bruta" es comparar todos contra todos, lo cual puede ser muy costoso ya que implica una comparación de todos contra todos. Es decir, si **`vectors`** tiene tamaño n entonce el número de comparaciones es n^2. Para evitar esto una posibilidad es usar **hashing**. Esta técnica consiste en tomar alguna función de los vectores tal que sea muy poco probable que dos vectores tengan la misma función. Por ejemplo en el caso anterior podemos tomar la suma de los elementos del vector. Si dos vectores **`v1,v2`** son iguales entonces su suma será ciertamente igual, aunque la recíproca no es cierto. Por ejemplo los vectores **`[0,1]`** y **`[1,0]`** tienen la misma suma pero son distintos. Pero si los elementos contenidos en los vectores son grandes y aleatorios, y la cantidad de elementos es grande la probabilidad de que tengan la misma suma es baja. Entonces podemos usar este truco simple de comparar la suma para descartar muchos pares de vectores.
A estas funciones que nos dan un entero a partir de un objeto, y que nos permiten identificar elementos iguales se les llama **funciones de hash**. En este ejercicio vamos a implementar a partir diferentes tipos de hashers como clases derivadas de la clase virtual pura

```
class inthasher_t {
public:
  virtual void reset()=0;
  virtual void hash(int x)=0;
  virtual int val()=0;
  virtual ~inthasher_t() {}
};
```

La idea es que, si tenemos un hasher **`h`**, la función **`h.reset()`** *reinicia* el hasher. Con **`h.hash(x)`** vamos hasheando los enteros, y cuando queremos el valor de hash de todo el objeto usamos **`h.val()`**. Entonces para hashear todo un arreglo de enteros podemos hacer

```
  vector<int> v = ...
  inthasher_t h;
  h.reset();
  for (auto x : v) h.hash(x);
  cout << "el hash de v es " << h.val() << endl;
  vector<int> v2 = ...
  h.reset();
  for (auto x : v2) h.hash(x);
  cout << "el hash de v2 es " << h.val() << endl;
```

[**sumhasher**]: Implementar una clase **`sumhasher_t`** derivada de **`inthasher_t`** que simplemente calcula la suma de lo enteros hasheados. Para eso tiene un estado interno que es un entero, y simplemente va sumando los enteros que son hasheados. La función **`reset()`** simplemente pone el estado interno en 0 y **`h.val()`** retorna el estado interno del hasher. Al aplicárselo a vectores tendremos,

```
v=[5,4,3] => 12
v=[8,3,5] => 16
```

[**cubhasher**]: Como vimos, el hash por la suma es muy débil ya que por ejemplo dos vectores que son uno una permutación del otro (por ej. **`[1,3,4]`** y **`[4,3,1]`**) tienen la misma suma 8. Entonces proponemos un hasher más complejo done el estado es actualizado con la expresión

$$s = (c_1 + c_2 s + x)^3, \quad c_1 = 56789234, c_2 = 12345678, \tag{1}$$

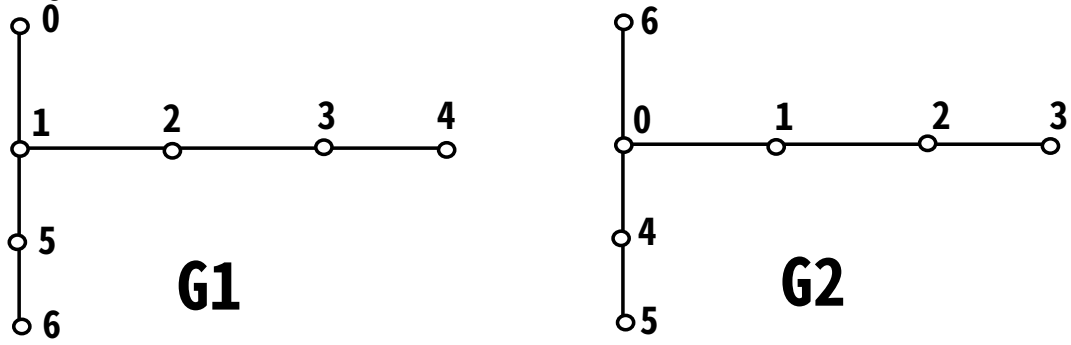
Los números $c_{1,2}$ son constantes completamente arbitrarias, tales que los números generados sean lo más *aleatorios* posibles. También el elevar al cubo, se podría reemplazar por otra potencia. Notar que el resultado de estas expresiones puede dar números mayores que el entero más grande representable con enteros de 32 bits ($O(10^9)$), pero no hay problema. Al hacer la potencia se produce un *overflow de enteros* y puede ser que se obtengan números negativos. No importa, lo importante es que el resultado sea **determinístico** y, como ya dijimos, que los números obtenidos estén bien distribuidos sobre los números enteros.

```
[5,4,3] => -444016179
[8,3,5] => -1949680043
```

[**count-different**] Escribir una función **`int count_different(vector<vector<int» &vectors, inthasher_t &hasher);`** que retorna la cantidad de vectores diferentes en **`vectors`**. Para ello calcula el hash de cada uno de los vectores y los va metiendo en un **`set<int>`**, la cantidad de vectores diferentes va a ser la cantidad de hashes guardados en el **`set`**.

Nota: En realidad podría haber vectores distintos que dan el mismo hash, pero asumimos que la función de hash es suficientemente *fuerte* como para que no se produzcan estas *colisiones* (dos objetos distintos que tienen el mismo hash).

[**isograph**] Sean dos grafos **`G1,G2`** queremos determinar si son **`isomorfos`** entre sí, es decir si hay una permutación en la numeración de los vértices que hace que los dos sean iguales, por ejemplo los dos grafos siguientes son isomorfos.



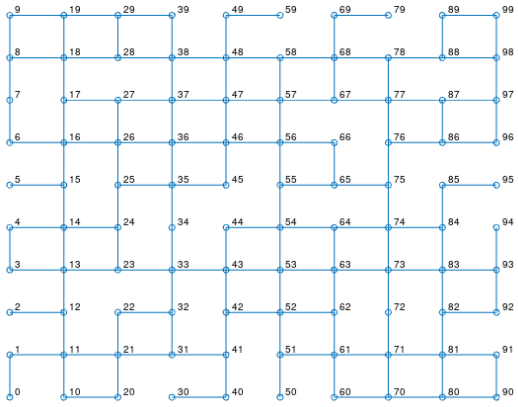
Consigna: Escribir una función
`void isograph(graph_t &g1,graph_t &g2,inthasher_t &hasher,vector<int> &perm);`
que determina la permutación que lleva el grafo **g1** al **g2**.

Los grafos son representados como `typedef vector<set<int» graph_t;`

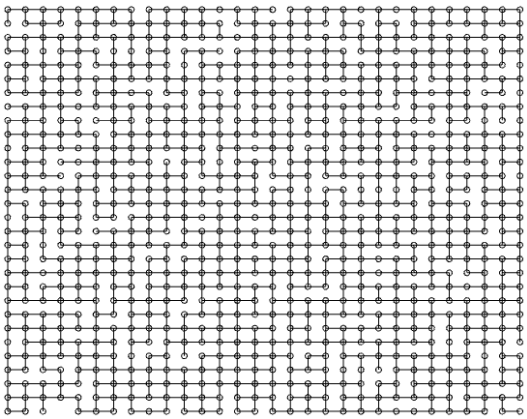
Ayuda: El algoritmo se basa en producir un para cada vértice del grafo, que identifica la varias propiedades del vértice, a saber su valencia, la valencia de sus vecinos, la de los vecinos de los vecinos... y así siguiendo. De manera que el hash del vértices es único. El algoritmo es así,

- Definir un vector de hashes `vector<int> hashv(nv);` donde **nv** es la cantidad de vértices.
- Inicializamos lo hashes a 0.
- Realizamos una serie de iteraciones en las cuales vamos reemplazando los hashes actuales por nuevos hashes, de la siguiente forma. Para cada vértices consruimos un conjunto con su valencia y los hashes de los vecinos. Por ejemplo para el vértice 1 en **G1** tenemos que construir el conjunto `{3,h(0),h(2),h(5)}`.
- Hacemos un hash de todos los elementos de ese conjunto y lo guardamos en un vector temporario `vector<int> hashvnew(nv);`
- Una vez que calculamos los nuevos hashes de los vértices, reemplazamos los viejos por los nuevos.
- Volvemos a repetir este proceso un cierto número de iteraciones, por ejemplo `niter=3*nv`. El número de iteraciones tiene que ser suficientemente grande como para que la información de los vértices se propague por todo el grafo, por eso debe ser al menos **nv**.
- Estos hashes deberían ser iguales para ambos grafos, es decir el hash obtenido par el vértice 0 en **G1** debería ser igual al del vértice 6 en **G2**. Entonces para encontrar la permutación podemos recorrer los vértices **j1** de **G1**, miramos su hash y lo buscamos entre los hashes de **G2**. El vértice de **j2** de **G2** que tiene ese hash es el que debemos aparear con **j1**, es decir `perm[j1]=j2`.

Nota: Usando este algoritmo se puede encontrar en forma muy eficiente la permutación entre grafos isomorfos muy grandes. El grafo **G(100)** del caso Nro 12 en **isograph.json** tiene 100 vértices y está representado abajo. A la derecha vemos un grafo **G(900)**con 900 vértices que también se puede numerar en segundos con este algoritmo.



G(100)



G(900)

El grafo de 100 vértices tiene,

```
 9 vértices de valencia 1
26 vértices de valencia 2
```

```
35 vértices de valencia 3
30 vértices de valencia 4
```

Si usáramos un algoritmo por fuerza bruta deberíamos hacer $100!\ O(10^{158})$ combinaciones. Si usamos el hecho de que las valencias son invariantes, entonces los vértices de valencia 1 en **G1** sólo pueden corresponder a los vértices de valencia 1 en **G2** por lo tanto sólo hay que probar $9!$ combinaciones para los vértices de valencia 1. Pero hay que hacer las combinaciones de los nodos de valencia 2, 3, y 4, de manera que el número total de combinaciones es

$$9!\,26!\,35!\,30! \sim O(4 \times 10^{104}). \tag{2}$$

Es decir que si bien el número de combinaciones baja muchísimo (50 órdenes de magnitud), de todas formas es imposible de calcular.

El costo del algoritmo usando hashing es $O(n^2 v)$, donde v es la valencia media del grafo. Esto se debe a que básicamente estamos haciendo $3n$ iteraciones (el número se puede ajustar), y el costo de cada iteración es $O(nv)$. En el caso de este grafo de 100 vértices presentado previamente, el número de valencias está acotado en 4 y en promedio es 2, de manera que el costo es $O(n^2)$.