

Reconstruction

Limite de temps : 4 secondes

Limite de mémoire : 1024 Mio

Ce problème est interactif.

Bissy est sur le point de participer au jeu d'orientation *Destination Finale*, organisé par des bénévoles, où les délégations EJOI vont faire une compétition pour visiter différents endroits à travers la ville. Mais Bissy s'intéresse plus à *comment* le jeu a été conçu qu'à le remporter.

Il y a N emplacements sur la carte et N équipes. Chaque équipe doit visiter tous les emplacements, et les emplacements de départ uniques — l'équipe i commence à l'emplacement i . Chaque équipe reçoit son propre itinéraire.

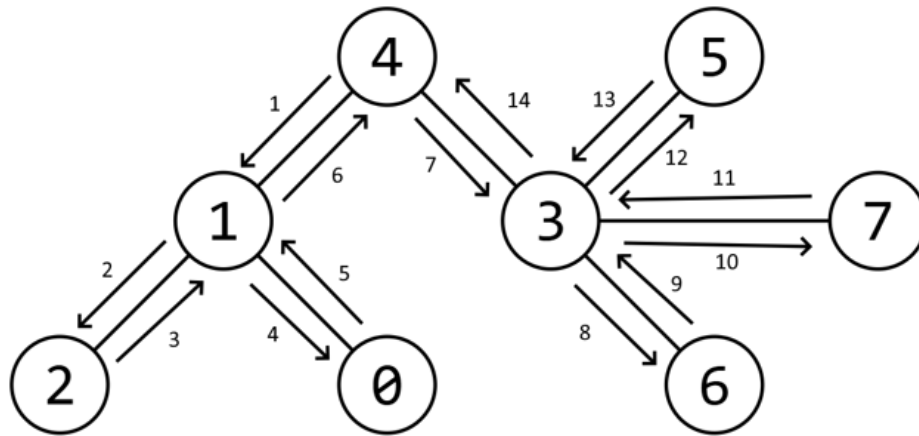
Bissy dispose aussi de quelques informations confidentielles — les itinéraires sont basés sur une structure cachée d'un réseau de lignes de transport public. Plus précisément, le jury du concours a choisi $N - 1$ lignes, chacune reliant une paire de lieux, de telle sorte que chaque lieu soit accessible depuis n'importe quel autre en utilisant ces lignes. Une telle structure est aussi appelée un arbre. Ensuite, pour chaque lieu de départ i , le jury a généré un *parcours DFS arbitraire* (défini ci-dessous) et a attribué ce parcours comme itinéraire à l'équipe i .

Bissy veut découvrir l'arbre de transport public que le jury a sélectionné. Elle ne pose que des questions rapides du type :

- Quelle est la j -ième destination de la i -ème équipe ?

Aidez Bissy en écrivant un programme `find_tree` qui trouve l'arbre caché.

Un *parcours DFS* d'un arbre est l'ordre dans lequel ses sommets sont visités pour la première fois par une procédure de parcours en profondeur. Une telle procédure commence à un sommet donné v et se déplace récursivement vers l'un de ses voisins non visités jusqu'à ce qu'aucun voisin de ce type n'existe — puis elle revient au sommet précédemment visité et continue à partir de là. Exemple :



Ici $v = 4$ et les flèches le long des arêtes correspondent aux étapes d'un parcours en profondeur. Le chemin généré est $[4, 1, 2, 0, 3, 6, 7, 5]$. Notez que l'ordre dans lequel les voisins sont visités est important ; un autre chemin pourrait être $[4, 3, 5, 7, 6, 1, 0, 2]$.

La structure de l'arbre et les N *parcours DFS* sont fixés avant le démarrage de votre programme et ne changent pas en fonction de vos questions. Les ordres des voisins des sommets utilisés lors de ces *parcours DFS* peuvent être différents.

Détails d'implémentation

Vous devez implémenter la fonction suivante :

```
std::vector<std::pair<int, int>> find_tree(int N)
```

- ici N est le nombre de lieux ;
- la valeur de retour est un tableau des arêtes de l'arbre — l'ordre des arêtes et leurs extrémités n'ont pas d'importance.

Pour chaque test, cette fonction peut être appelée jusqu'à T fois.

Pour interagir avec le jury, vous pouvez faire des appels à la fonction suivante :

```
int guess(int i, int j)
```

Elle retourne la j -ème position dans le i -ème parcours de l'arbre. Notez que `guess(i, 0)` retournera i . Vous pouvez supposer que la fonction répond en temps constant $O(1)$ pour toutes les sous-tâches jusqu'à 6, et en temps logarithmique $O(\log N)$ pour la sous-tâche 7. Vous devez faire en sorte que $0 \leq i, j \leq N - 1$, sinon votre solution recevra le verdict `Output isn't correct: Invalid call`.

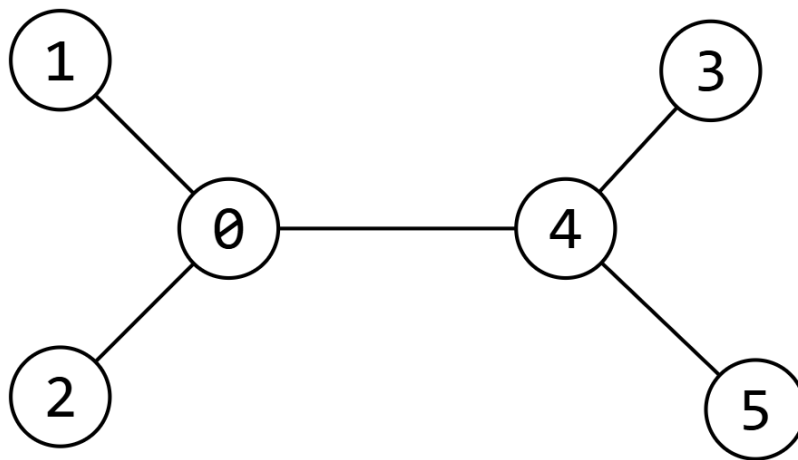
Contraintes

- $2 \leq N \leq 2^{16} + 1$
- T vérifie les contraintes suivantes en fonction du N maximum (noté N_{max}) parmi les appels de fonction dans un test :
 - si $N_{max} \leq 9$, alors $1 \leq T \leq 100$
 - si $N_{max} \leq 2^{10} + 1$, alors $1 \leq T \leq 10$
 - si $N_{max} \leq 2^{16} + 1$, alors $1 \leq T \leq 3$

L'évaluateur peut utiliser jusqu'à 280 Mio, qui comptent dans la limite mémoire de votre solution.

Exemple

Supposez que la structure cachée du réseau de lignes de transport public est la suivante :



Supposons que pour la position de départ 0, le parcours soit $[0, 1, 2, 4, 3, 5]$. Voici un exemple d'interaction :

Programme du participant	Programme du jury
	<code>find_tree(6)</code>
<code>guess(0, 0)</code>	0
<code>guess(0, 1)</code>	1
<code>guess(0, 2)</code>	2
<code>guess(0, 3)</code>	4
<code>guess(0, 4)</code>	3
<code>guess(0, 5)</code>	5
<code>return {{0,1},{0,2},{4,0},{5,4},{3,4}};</code>	

Notez que les informations provenant des requêtes ne suffisent pas à déterminer l'arbre de manière unique, mais la réponse correspond au test dans la sous-tâche 0.

Sous-tâches

Sous-tâche	Points	N	Contraintes supplémentaires
0	0	-	Exemple du sujet.
1	11	≤ 9	-
2	6	≤ 100	Chaque sommet est connecté à au maximum 2 autres.
3	13	≤ 100	Chaque chemin est généré par une recherche en profondeur qui, à chaque étape, priorise l'éloignement du sommet 0. S'il existe plusieurs options pour s'éloigner du sommet 0, l'une d'elles est choisie arbitrairement.
4	11	≤ 100	Tout sommet autre que 0 est connecté à au maximum 2 autres.
5	10	≤ 100	-
6	31	$\leq 2^{10} + 1$	-
7	18	$\leq 2^{16} + 1$	-

Score

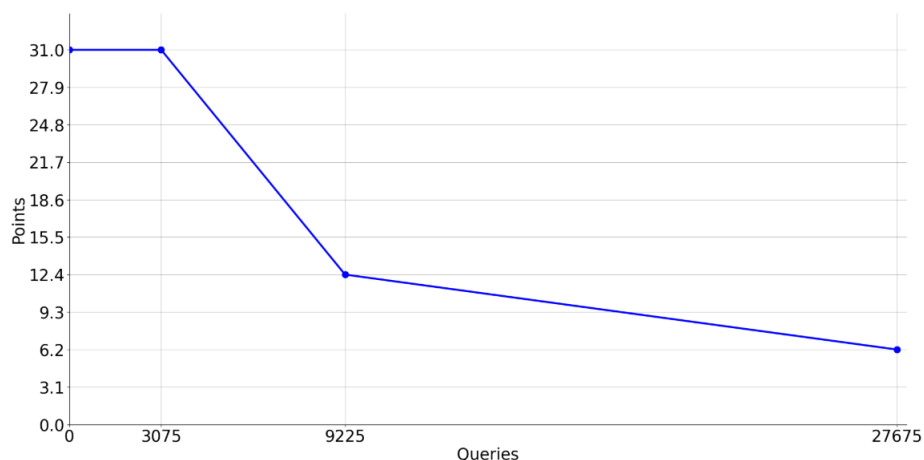
Pour chaque sous-tâche jusqu'à la sous-tâche 5, vous obtenez la totalité des points si vous trouvez avec succès l'arbre dans les limites de temps imposées. Pour les sous-tâches 6 et 7, la fraction des points totaux S pour un test donné dépend du nombre **maximum** de requêtes que vous posez sur l'un de ses sous-tests, Q_{max} . Le calcul des points pour ces sous-tâches est le suivant :

- si $Q_{max} \leq L_1$, alors $S = 1.0$;
- si $L_1 < Q_{max} \leq L_2$, alors $S = 0.4 + 0.6 \times \frac{L_2 - Q_{max}}{L_2 - L_1}$;
- si $L_2 < Q_{max} \leq 3L_2$, alors $S = 0.2 + 0.2 \times \frac{3L_2 - Q_{max}}{2L_2}$;
- si $3L_2 < Q_{max}$, alors $S = 0.2$;

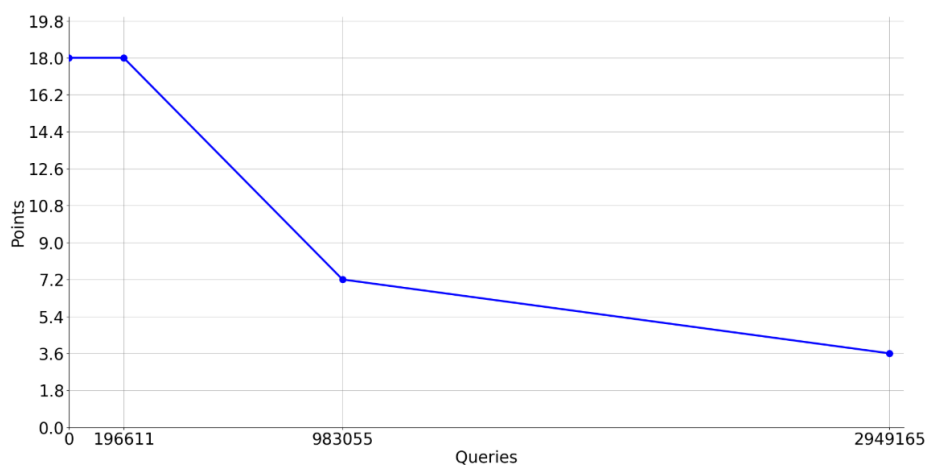
où $L_1 = 3 \times (2^{10} + 1) = 3075$ et $L_2 = 9 \times (2^{10} + 1) = 9225$ pour la sous-tâche 6, et $L_1 = 3 \times (2^{16} + 1) = 196611$ et $L_2 = 15 \times (2^{16} + 1) = 983055$ pour la sous-tâche 7. La fraction de points que vous obtenez pour l'ensemble de la sous-tâche est le minimum S parmi tous les tests.

Les graphiques ci-dessous montrent la fonction de calcul des points pour les deux dernières sous-tâches :

Sous-tâche 6



Sous-tâche 7



Évaluateur d'exemple

Il y a deux évaluateurs d'exemple fournis.

Pour les tests locaux : `Lgrader.cpp` peut être compilé avec votre programme pour fournir un outil de test. Le programme lit le nombre T de sous-tests et pour chacun d'eux attend le nombre d'emplacements N suivi de $N - 1$ lignes, chacune avec une paire d'entiers décrivant une ligne de transport. Après cela, N lignes de N entiers suivent – les parcours DFS. Notez que le parcours i doit commencer à partir du sommet i . Vous pouvez laisser l'évaluateur générer les parcours en mettant la constante `AUTO_GENERATE` à `true`. En sortie, l'évaluateur donne soit un message d'erreur si le résultat est incorrect, soit le nombre de requêtes posées pour chaque sous-test et leur maximum. Notez que l'évaluateur ne fonctionne pas pour des valeurs de N trop grandes (plus précisément pour la sous-tâche 7), et vous êtes libre d'utiliser votre imagination pour ce cas.

Pour les tests utilisateur sur le système : `stub.cpp` peut être utilisé avec votre programme pour les tests utilisateur. Le format d'entrée est le même que celui de l'autre évaluateur. Notez que pour les tests utilisateur, il n'y a pas d'option intégrée pour la génération automatique de parcours.