

Rekonstruieren

Zeit: 4 Sekunden

Speicherplatz: 1024 MiB

Dies ist eine interaktive Aufgabe.

Bissy ist kurz davor, am von Freiwilligen organisierten Orientierungsspiel *Final destination* teilzunehmen. Bei diesem Spiel werden EJOI-Delegationen um die Wette verschiedene Orte der Stadt aufsuchen. Bissy ist allerdings mehr daran interessiert, *wie* das Spiel entworfen wurde, als daran, es zu gewinnen.

Es gibt N Orte auf der Karte und N Teams. Jedes Team hat einen eindeutigen Startort und muss jeden anderen Ort aufsuchen — Team i startet an Ort i . Jedes Team hat eine eigene Route.

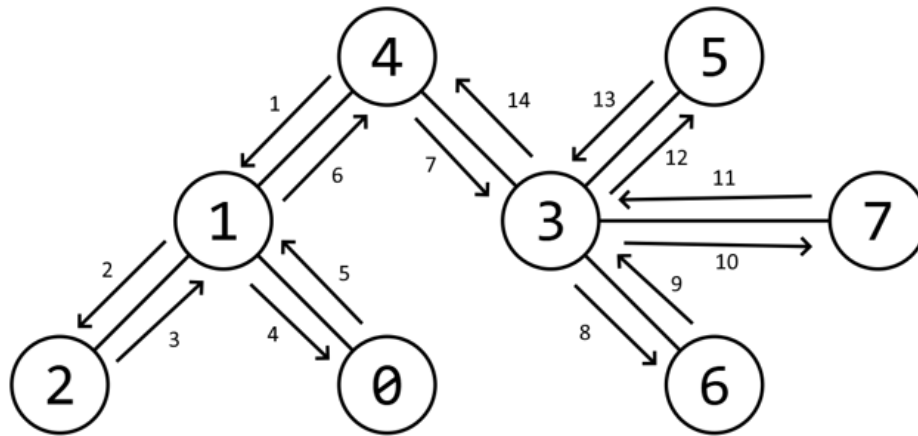
Bissy hat außerdem einige Insiderinformationen — die Routen basieren auf einer versteckten Struktur von öffentlichen Nahverkehrslinien. Genauer gesagt hat die Jury des Orientierungsspiels $N - 1$ Linien gewählt, von denen jede zwei Orte verbindet, sodass jeder Ort von jedem anderen Ort aus mit diesen Linien erreichbar ist. Eine solche Struktur nennen wir Baum. Für jeden Startort i hat die Jury dann einen **zufälligen** *DFS walk* (Definition unten) generiert und diesen *DFS walk* einem Team i als Route zugewiesen.

Bissy möchte herausfinden, welchen Baum die Jury als Grundlage gewählt hat. Dafür stellt sie ausschließlich kurze Fragen der folgenden Art:

- Was ist die j -te Station des i -ten Teams?

Hilf Bissy und schreibe ein Programm `find_tree`, welches den versteckten Baum findet.

Ein *DFS walk* eines Baumes ist die Reihenfolge, in der die Knoten des Baums zum ersten Mal durch eine Tiefensuche besucht werden. So ein Verfahren startet an einem gegebenen Knoten v und geht rekursiv zu einem seiner unbesuchten Nachbarn, bis es keinen unbesuchten Nachbarn mehr gibt — dann geht es zum zuvor besuchten Knoten zurück und geht von dort aus weiter. Beispiel:



Hier ist $v = 4$ und die Pfeile entlang der Kanten entsprechen den Schritten einer Tiefensuche. Der *DFS walk* ist $[4, 1, 2, 0, 3, 6, 7, 5]$. Beachte, dass die Reihenfolge, in der die Nachbarn besucht werden, wichtig ist; ein anderer *walk* wäre $[4, 3, 5, 7, 6, 1, 0, 2]$.

Die Baumstruktur und die N *DFS walks* stehen fest, bevor dein Programm startet, und ändern sich nicht in Reaktion auf deine Fragen. Die Reihenfolgen der Nachbarn der Knoten, die während dieser Tiefensuchen verwendet werden, können unterschiedlich sein.

Implementierungsdetails

Du musst die folgende Funktion implementieren:

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

- N ist die Anzahl an Orten;
- Der Rückgabewert ist eine Liste der Kanten des Baums — die Reihenfolge der Kanten und ihrer Endpunkte sind dabei egal.

Für jeden Test kann die Funktion bis zu T Mal aufgerufen werden.

Um mit der Jury zu interagieren, kannst du die folgende Funktion aufrufen:

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

Diese Funktion hat als Rückgabewert den j -ten Ort des i -ten *DFS walks* des Baums. Beachte, dass `guess( $i, 0$ )` den Wert i zurück gibt. Du darfst davon ausgehen, dass die Funktion in konstanter Zeit $O(1)$ für alle Teilaufgaben bis Teilaufgabe 6 antwortet und in logarithmischer Zeit $O(\log N)$ für Teilaufgabe 7. Es muss $0 \leq i, j \leq N - 1$ gelten, sonst wird deine Lösung das Ergebnis `Output isn't correct: Invalid call` erhalten.

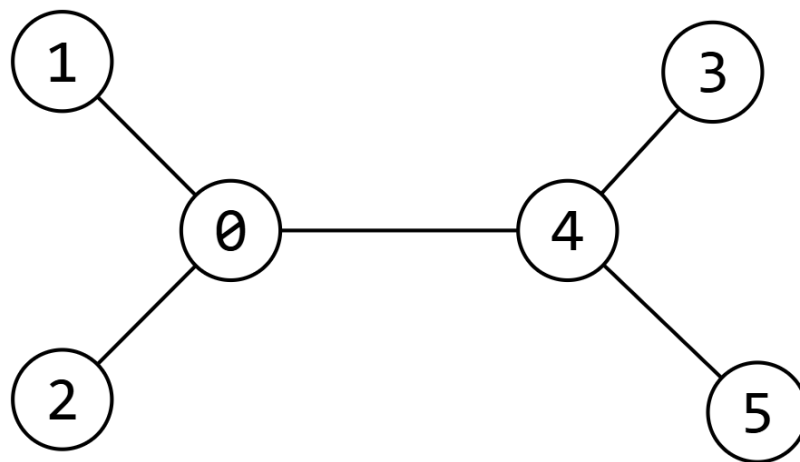
Beschränkungen

- $2 \leq N \leq 2^{16} + 1$
- T folgt den Beschränkungen basierend auf dem maximalen Wert von N (im Folgenden bezeichnet als N_{max}) unter den Funktionsaufrufen in einem Test:
 - falls $N_{max} \leq 9$, dann $1 \leq T \leq 100$
 - falls $N_{max} \leq 2^{10} + 1$, dann $1 \leq T \leq 10$
 - falls $N_{max} \leq 2^{16} + 1$, dann $1 \leq T \leq 3$

Der System Grader kann bis zu 280 MiB Speicherplatz benötigen. Dies zählt zusätzlich zu dem Speicherverbrauch deiner Lösung.

Beispiele

Nimm an, dass der versteckte Baum wie folgt aussieht:



Nimm ebenfalls an, dass $[0, 1, 2, 4, 3, 5]$ der *DFS walk* für Startort 0 ist. Dann ist dies eine beispielhafte Interaktion mit der Jury:

Dein Programm	Programm der 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>	

Beachte, dass die Informationen aus den Anfragen nicht ausreichen, um den Baum eindeutig zu bestimmen. Nichtsdestotrotz ist dies die Antwort auf den Test in Teilaufgabe 0.

Teilaufgaben

Teilaufgabe	Punkte	N	Zusätzliche Beschränkungen
0	0	-	Das Beispiel.
1	11	≤ 9	-
2	6	≤ 100	Jeder Knoten ist mit maximal 2 anderen verbunden.
3	13	≤ 100	Jeder <i>walk</i> wird durch eine Tiefensuche erzeugt, die sich bei jedem Schritt von Knoten 0 entfernt, wenn möglich. Falls es mehrere Möglichkeiten gibt, sich von Knoten 0 zu entfernen, wird eine zufällige ausgewählt.
4	11	≤ 100	Jeder Knoten, außer Knoten 0, ist mit maximal 2 anderen verbunden.
5	10	≤ 100	-
6	31	$\leq 2^{10} + 1$	-
7	18	$\leq 2^{16} + 1$	-

Punkte

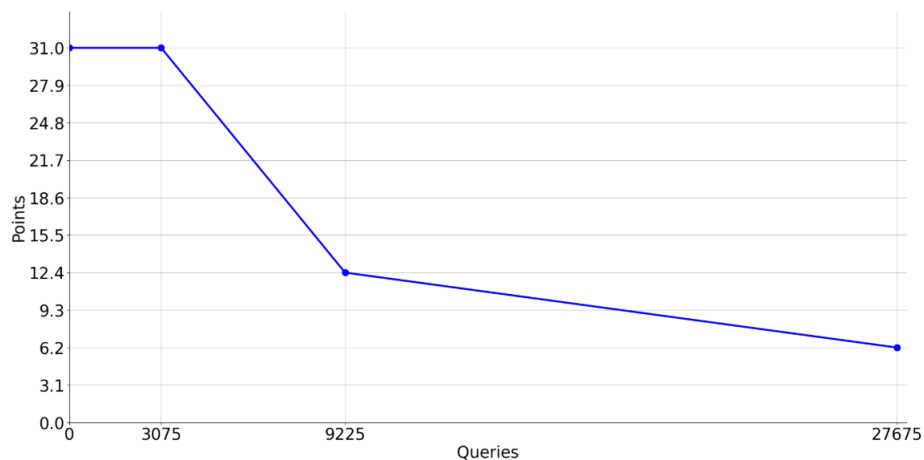
Für jede Teilaufgabe bis Teilaufgabe 5 erhältst du die volle Punktzahl, wenn du innerhalb des Zeitlimits erfolgreich den Baum findest. Für Teilaufgaben 6 und 7 hängt der Anteil der vollen Punkte S für einen gegebenen Test von der **maximalen** Anzahl von Anfragen Q_{max} ab, die du bei einem der Untertests stellst. Die Bewertung für diese Teilaufgaben ist wie folgt:

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

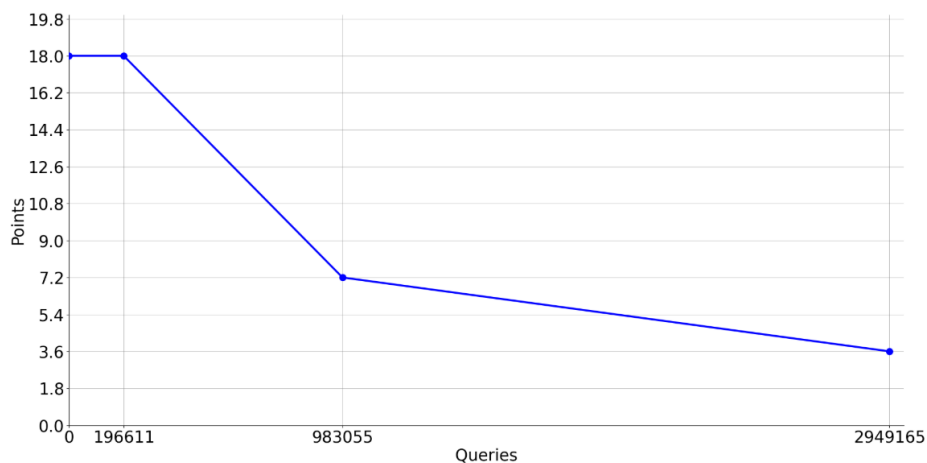
mit $L_1 = 3 \times (2^{10} + 1) = 3075$ und $L_2 = 9 \times (2^{10} + 1) = 9225$ für Teilaufgabe 6, sowie $L_1 = 3 \times (2^{16} + 1) = 196611$ und $L_2 = 15 \times (2^{16} + 1) = 983055$ für Teilaufgabe 7. Der Anteil an Punkten die du für die komplette Teilaufgabe erhältst ist das Minimum S aller Tests.

Die folgenden Graphen zeigen die Punkte-Funktion der letzten beiden Teilaufgaben:

Subtask 6



Subtask 7



Sample Grader

Es werden zwei Sample Grader zur Verfügung gestellt.

Zum lokalen Testen: `Lgrader.cpp` kann zusammen mit deinem Programm kompiliert werden und ist ein Werkzeug zum Testen. Das Programm liest die Anzahl T der Testfälle und erwartet für jeden davon die Anzahl der Orte N , gefolgt von $N - 1$ Zeilen, die jeweils ein Paar von ganzen Zahlen enthalten, das eine direkte Verbindung beschreibt. Danach folgen N ganze Zahlen – die *DFS walks*. Beachte, dass *walk* i am Knoten i starten muss. Du kannst den Grader *DFS walks* generieren lassen, indem du `AUTO_GENERATE` auf `true` setzt. Als Ausgabe meldet der Grader entweder eine Fehlermeldung, wenn das Ergebnis falsch ist, oder die Anzahl der Anfragen, die für jeden Testfall gestellt wurden, sowie ein Gesamtmaximum. Beachte außerdem, dass der Grader nicht für große N funktioniert (genauer gesagt für die Beschränkungen in Teilaufgabe 7). Es steht dir frei, hier deiner Kreativität freien Lauf zu lassen.

Für System-Benutzertests: `stub.cpp` kann zusammen mit deinem Programm für die Benutzertests auf dem System verwendet werden. Das Eingabeformat ist dasselbe wie das für den anderen

Grader. Beachte, dass es für Benutzertests keine eingebaute Option zur automatischen Erzeugung von Wegen gibt.