რეკონსტრუქცია

დროის ლიმიტი: 4 წამი
მეხსიერების ლიმიტი: 1024 MiB

ეს არის ინტერაქტიული ამოცანა.

ბისი აპირებს მონაწილეობა მიიღოს მოხალისეების მიერ ორგანიზებულ თამაშში *საბოლოო დანიშნულება*, სადაც EJOI დელეგაციები ეჯიბრებიან ერთმანეთს სხვადსხვა ლოკაციებზე მისვლაში. მაგრამ ბისი თამაშის დიზაინით უფროა დაინტერესებული ვიდრე მის მოგებით.

რუკაზე არის N ლოკაცია და N გუნდი. ყველა გუნდი უნდა ეწვიოს ყველა ლოკაციას და თითოეულ მათგანს აქვს უნიკალური საწყისი ლოკაცა — გუნდი i იწყებს ლოკაციიდან i. ყველა გუნდს მიეცემა საკუთარი მარშრუტი.

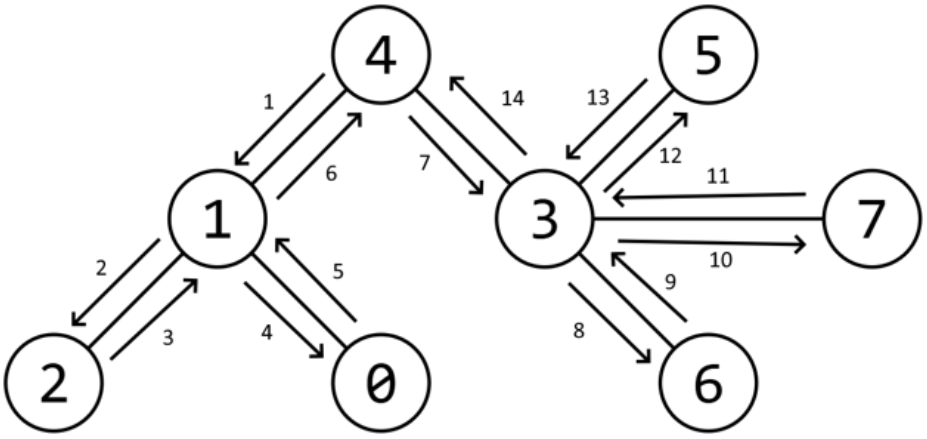
ბისის ასევე აქვს გარკვეული შიდა ინფორმაცია — მარშრუტები ეფუძნება სწრაფი საზოგადოებრივი ტრანსპორტის ხაზების ფარულ სტრუქტურას. უფრო ზუსტად, შეჯიბრის ჟიურიმ აირჩია $(N-1)$ რაოდენობის ხაზი, რომელთაგან თითოეული აკავშირებს ლოკაციათა წყვილს ისე, რომ ნებისმიერი ლოკაცია მიღწევადია ნებისმიერი სხვა ლოკაციიდან ამ ხაზების გამოყენებით. ასეთ სტრუქტურას ასევე ხე ეწოდება. შემდეგ, თითოეული საწყისი i ლოკაციისთვის ჟიურიმ **ნებისმიერად** დააგენერირა *DFS შემოვლა* (განსაზღვრულია ქვემოთ) და აირჩია ამ შემოვლის თანმიმდევრობა i გუნდის გასავლელ მარშრუტად.

ბისის სურს გაარკვიოს ის საზოგადოებრივი ტრანსპორტის ხე, რომელიც ჟიურიმ შეარჩია. ის სვამს მხოლოდ შემდეგი სახის კითხვებს:

- რომელია i-ური გუნდის მარშრუტში j-ური დანიშნულების ადგილი?

დაეხმარე ბისის და დაწერე პროგრამა `find_tree`, რომელიც იპოვის დამალულ ხეს.

> ხის *DFS შემოვლა* არის ის თანმიმდევრობა, რომლითაც მისი წვეროები პირველად მოინახულება სიღრმეში ძებნის (depth-first-search) პროცედურით. ასეთი პროცედურა იწყება მოცემული v წვეროდან და რეკურსიულად გადადის ერთ-ერთ ჯერ მოუნახულებელ მეზობელ წვეროზე, სანამ ასეთი მეზობელი არსებობს — შემდეგ ის ბრუნდება წინა მონახულებულ წვეროზე და იქიდან აგრძელებს ძებნას. მაგალითი:



აქ $v = 4$ და ისრები წიბოების გასწვრივ შეესაბამება სიღრმეში ძებნის (depth-first-search) ნაბიჯებს. დაგენერირებული შემოვლა არის `[4,1,2,0,3,6,7,5]`. გაითვალისწინეთ, რომ მნიშვნელობა აქვს იმ თანმიმდევრობას, რომლითაც მეზობლებს ვსტუმრობთ; სხვა დასაშვები შემოვლა შეიძლება იყოს `[4,3,5,7,6,1,0,2]`.

ხის სტრუქტურა ფიქსირებულია თქვენი პროგრამის დაწყებამდე და არ იცვლება თქვენი კითხვების მიხედვით.

იმპლემენტაციის დეტალები

თქვენ იმპლემენტაცია უნდა გაუკეთოთ შემდეგ ფუნქციას:

```
std::vector> find_tree(int N)
```

- აქ N არის ლოკაციების რაოდენობა;
- დასაბრუნებელი მნიშვნელობა არის ხის წიბოების სია — წიბოების თანმიმდევრობას და თითო წიბოსთვის წვეროების თანმიმდევრობას მნიშვნელობა არ აქვს.

თითოეული ტესტისთვის ეს ფუნქცია შეიძლება გამოძახებული იყოს T-ჯერ.

ჟიურისთან ურთიერთობისთვის შეგიძლიათ გამოიძახოთ შემდეგი ფუნქცია:

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

ის აბრუნებს ხის i-ური წვეროდან შემოვლაში რიგით j-ურ ლოკაციას. გაითვალისწინეთ, რომ `guess(i, 0)` დააბრუნებს i-ს. შეგიძლიათ ჩათვალოთ, რომ ფუნქცია პასუხობს მუდმივ დროში $O(1)$ ყველა ქვეამოცანისთვის მე-6 ქვეამოცანამდე და ლოგარითმულ დროში $O(\log N)$ ქვეამოცანა 7-ისათვის. აუცილებელია სრულდებოდეს $0 \leq i, j \leq N - 1$, წინააღმდეგ შემთხვევაში თქვენი ამოხსნა მიიღებს ვერდიქტს `Output isn't correct: Invalid call`.

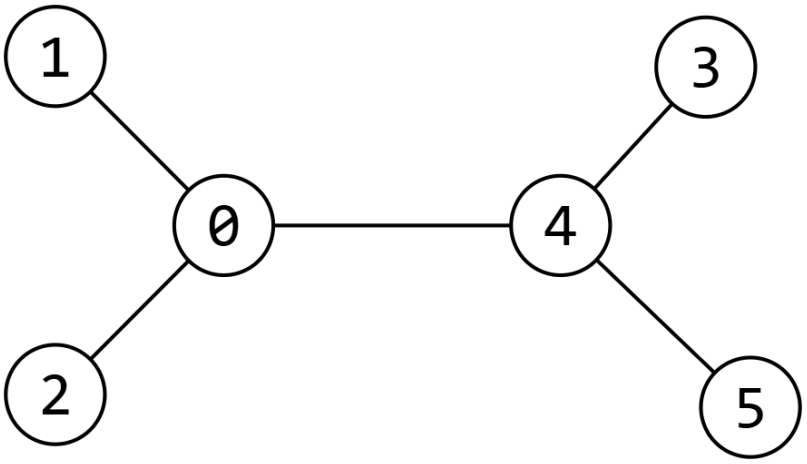
შეზღუდვები

- $2 \leq N \leq 2^{16} + 1$
- T აკმაყოფილებს შეზღუდვებს, რომლებიც დამოკიდებულია მაქსიმალურ N-ზე (აღინიშნება როგორც N_{max}) ერთი ტესტის ფარგლებში ფუნქციის გამოძახებებს შორის:
  - თუ $N_{max} \leq 9$, მაშინ $1 \leq T \leq 100$
  - თუ $N_{max} \leq 2^{10} + 1$, მაშინ $1 \leq T \leq 10$
  - თუ $N_{max} \leq 2^{16} + 1$, მაშინ $1 \leq T \leq 3$

სისტემის გრადერს შეუძლია 280 MB-მდე გამოყენება, რაც თქვენი ამოხსნის მეხსიერებაში ითვლება.

მაგალითი

დავუშვათ, სწრაფი საზოგადოებრივი ტრანსპორტის ხაზების ფარული სტრუქტურა შემდეგნაირია:



დავუშვათ, რომ საწყისი ლოკაცია 0-დან შემოვლის თანმიმდევრობა არის $[0, 1, 2, 4, 3, 5]$. მაშინ, ქვემოთ მოცემულია ინტერაქციის მაგალითი:

მონაწილის პროგრამა	ჟიურის პროგრამა
	`find_tree(6)`
`guess(0, 0)`	`0`
`guess(0, 1)`	`1`
`guess(0, 2)`	`2`
`guess(0, 3)`	`4`
`guess(0, 4)`	`3`
`guess(0, 5)`	`5`
`return {{0,1},{0,2},{4,0},{5,4},{3,4}};`	

გაითვალისწინეთ, რომ მოთხოვნებიდან მიღებული ინფორმაცია საკმარისი არ არის ხის ცალსახად დასადგენად, მაგრამ ეს სინამდვილეში არის პასუხი ტესტისთვის ქვეამოცანაში 0.

ქვეამოცანები

ქვეამოცანა	ქულები	N	დამატებითი შეზღუდვები
0	0	–	მაგალითი.
1	11	≤ 9	–
2	6	≤ 100	ყოველი წვერო დაკავშირებულია არაუმეტეს 2 სხვა წვეროსთან.
3	13	≤ 100	ყოველი შემოვლა გენერირებულია ისეთი სიღრმეში ძიების (depth-first-search) მიერ, რომელიც ყოველ ნაბიჯზე პრიორიტეტს ანიჭებს წვერო 0-დან დაშორების გაზრდას. თუ არსებობს წვერო 0-დან დაშორების გაზრდის რამდენიმე ვარიანტი, ერთ-ერთი მათგანი აირჩევა ნებისმიერად.
4	11	≤ 100	ყველა წვეროს 0-ის გარდა აქვს 3-ზე ნაკლები მეზობელი.
5	10	≤ 100	–
6	31	$\leq 2^{10}+1$	–
7	18	$\leq 2^{16}+1$	–

შეფასება

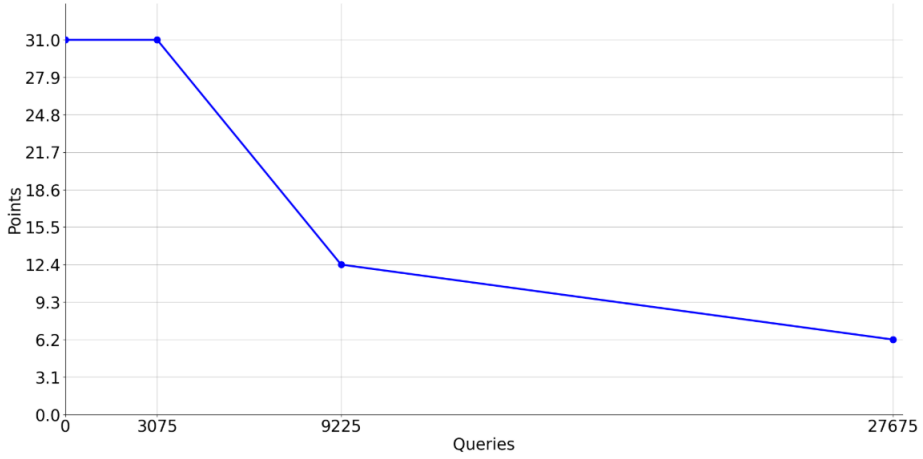
ყოველი ქვეამოცანისთვის 1-ლიდან მე-5 ქვეამოცანის ჩათვლით, თქვენ მიიღებთ სრულ ქულებს, თუ დროის ლიმიტის ფარგლებში წარმატებით იპოვით ხეს. ქვეამოცანებისთვის 6 და 7, სრული ქულების წილი S მოცემული ტესტისთვის დამოკიდებულია მისი ქვეტესტებისთვის თქვენ მიერ დასმული შეკითხვების რაოდენობებს შორის **მაქსიმალურზე**, Q_{max}. ამ ქვეამოცანების ქულების დათვლა შემდეგნაირად ხდება:

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

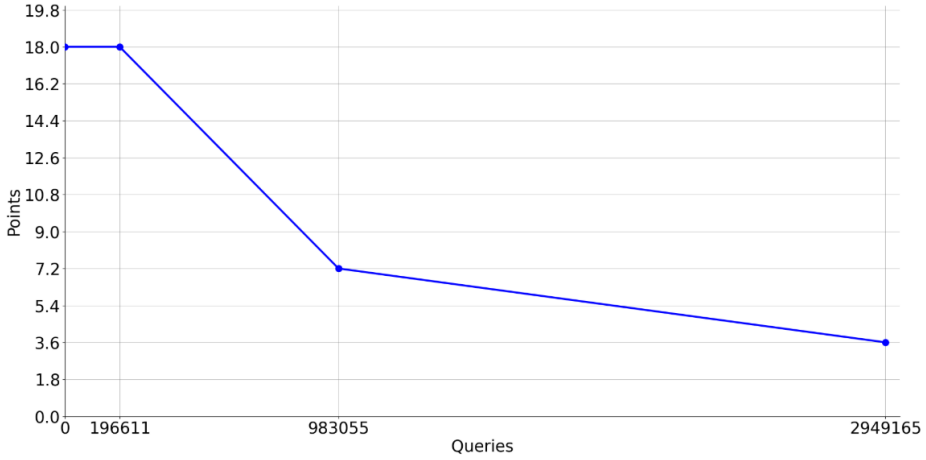
სადაც $L_1 = 3 \times (2^{10}+1) = 3075$ და $L_2 = 9 \times (2^{10}+1) = 9225$ ქვეამოცანა 6-ისთვის, ხოლო $L_1 = 3 \times (2^{16}+1) = 196611$ და $L_2 = 15 \times (2^{16}+1) = 983055$ ქვეამოცანა 7-ისთვის. ქულების წილი, რომელსაც მიიღებთ მთელი ქვეამოცანიდან, არის მინიმალური S მის ყველა ტესტს შორის.

ქვემოთ მოცემული გრაფიკები აჩვენებს ქულების დათვლის ფუნქციას ბოლო ორი ქვეამოცანისთვის:

ქვეამოცანა 6



ქვეამოცანა 7



სანიმუშო გრეიდერი

მოცემულია ორი სანიმუშო გრეიდერი.

ლოკალური ტესტირებისთვის: `Lgrader.cpp` შეგიძლიათ დააკომპილიროთ თქვენს პროგრამასთან ერთად, რომ მიიღოთ ტესტირების ინსტრუმენტი. პროგრამა კითხულობს ტესტში გამოძახებების რაოდენობას T და თითოეულისთვის შემოვლის ლოკაციების რაოდენობას N, რასაც მოსდევს $N-1$ სტრიქონი, თითოეული ორი მთელი რიცხვით, რომლებიც აღწერენ ორი ლოკაციის დამაკავშირებელ ხაზს. ამის შემდეგ მოსდევს N სტრიქონი თითოეული N მთელი რიცხვით – DFS შემოვლები. გაითვალისწინეთ, რომ შემოვლა i უნდა დაიწყოს წვეროდან i. თქვენ შეგიძლიათ ნება დართოთ შემფასებელს გენერირება გაუკეთოს შემოვლებს თავისით, თუ დააყენებთ კონსტანტას `AUTO_GENERATE` მნიშვნელობაზე `true`. შედეგის გამოტანისას გრადერი ან შეგატყობინებთ შეცდომის შესახებ, თუ შედეგი არასწორია, ან დააბრუნებს თითოეული გამოძახებისთვის დასმული შეკითხვების რაოდენობას და მათ შორის მაქსიმუმს. ასევე გაითვალისწინეთ, რომ შემფასებელი არ მუშაობს საკმარისად დიდი N-სთვის (უფრო ზუსტად,

ქვეამოცანა 7-ის შეზღუდვები), ამ შემთხვევისთვის თავისუფლად შეგიძლიათ გამოიყენოთ თქვენი ფანტაზია.

სისტემაში მომხმარებლის ტესტებისთვის: `stub.cpp` შეგიძლიათ გამოიყენოთ თქვენს პროგრამასთან ერთად სისტემაში მომხმარებლის ტესტებისთვის. შესატანი მონაცემების ფორმატი იგივეა, რაც სხვა გრადერისთვის. გაითვალისწინეთ, რომ მომხმარებლის ტესტებისთვის არ არსებობს ჩაშენებული ვარიანტი შემოვლების ავტომატური გენერაციისთვის.