Solutions PO camp 2026 day 2

1 Fredrik’s Pizzeria

Fredrik's Pizzeria

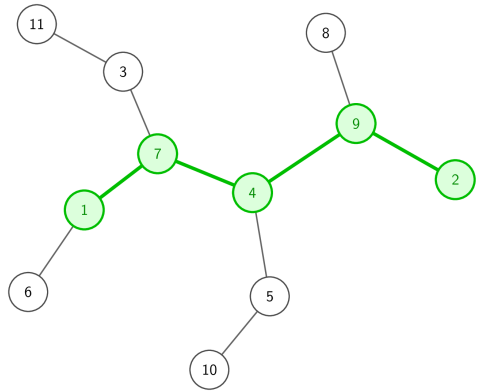
Author: Joshua Andersson

Description

You are given an undirected *cactus* graph (every edge is part of at most one simple cycle). We can remove edges. Every edge is *linked* to another one, its *partner*. If you remove an edge, its partner is also removed. Can you remove edges such that nodes 1 and 2 are in the same component, but 1 and 3 are in different components?

Subtask 1 and 2: The graph is a tree

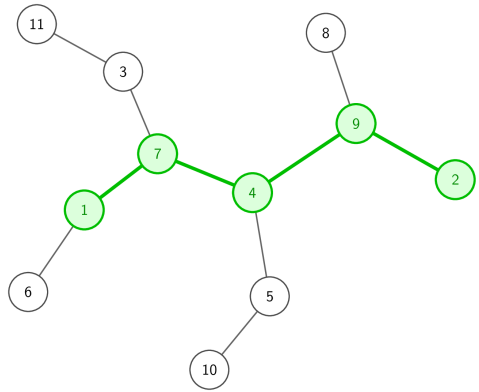
Because the graph is a tree, there exists exactly one path between vertices 1 and 2. We might as well take it: any deviation only helps the rat.



Subtask 1 and 2: The graph is a tree (ii)

Because the graph is a tree, there exists exactly one path between vertices 1 and 2. We might as well take it: any deviation only helps the rat.

We can greedily choose to keep only this path and the edges linked to edges on the path.

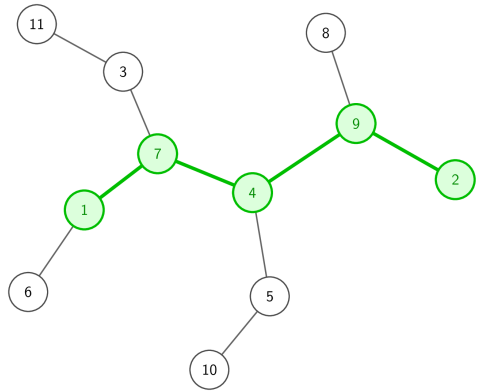


Subtask 1 and 2: The graph is a tree (iii)

Because the graph is a tree, there exists exactly one path between vertices 1 and 2. We might as well take it: any deviation only helps the rat.

We can greedily choose to keep only this path and the edges linked to edges on the path.

This is optimal: keeping any other edges doesn't help, and only helps with connecting 1 and 3.



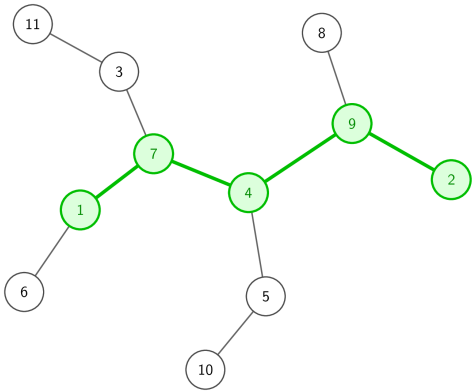
Subtask 1 and 2: The graph is a tree (iv)

Because the graph is a tree, there exists exactly one path between vertices 1 and 2. We might as well take it: any deviation only helps the rat.

We can greedily choose to keep only this path and the edges linked to edges on the path.

This is optimal: keeping any other edges doesn't help, and only helps with connecting 1 and 3.

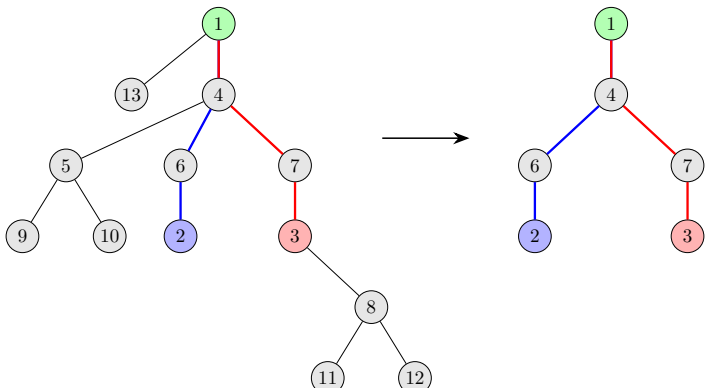
After removing all edges not on the path or linked to it, check if 1 and 3 are connected. This runs in $O(N)$.



Subtask 1 and 2: Both paths at once

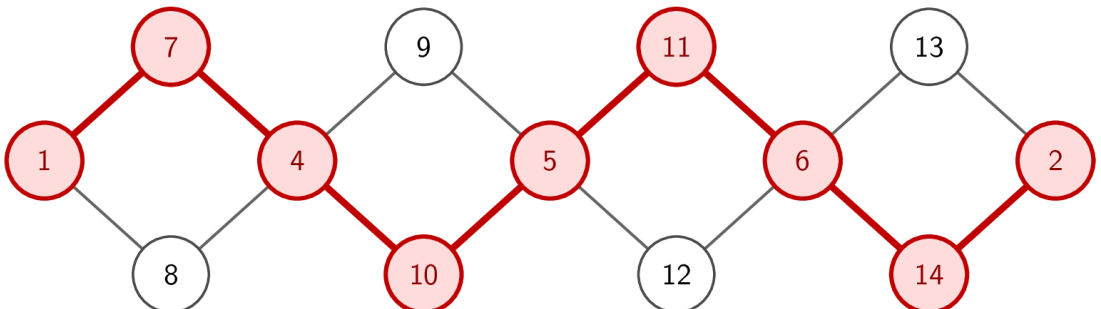
Another way to see it: only the paths $1 \to 2$ (blue) and $1 \to 3$ (red) matter. Every edge outside them can be pruned. Since the path $1 \to 2$ is now forced, we can add all edges linked to it and check whether that connected the rat.

Keep the shape of the pruned tree in mind: it is a Y. This shape will come back in the full solution.



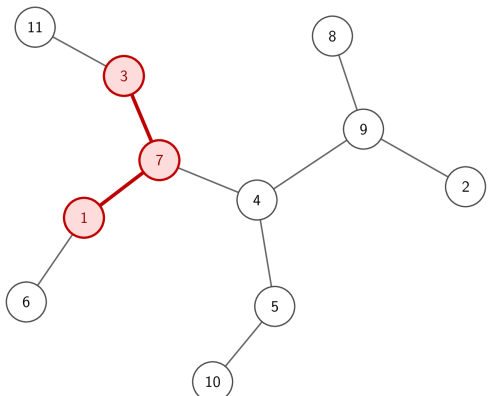
Generalizing to a cactus

Unfortunately, a cactus can have an exponential number of paths between two vertices, so the idea of guessing a path and trying it doesn't generalize. The following graph has an exponential number of paths between vertex 1 and 2.



Generalizing to a cactus: second try

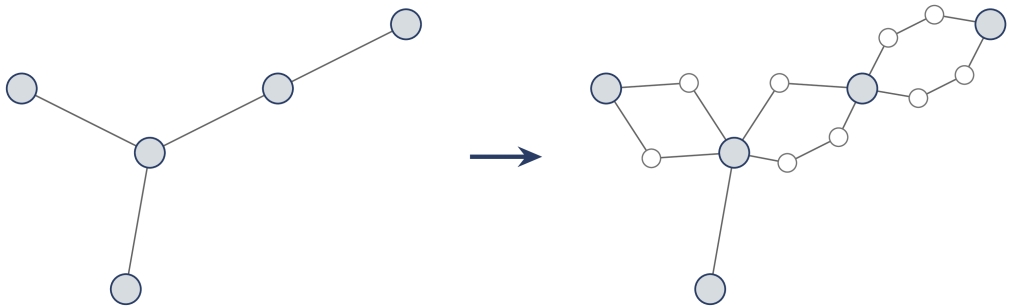
There exists another way to solve the tree case that is slightly harder, but generalizes to a cactus. Instead of focusing on trying to connect 1 and 2, focus on disconnecting 1 and 3. In a tree, it is always enough to remove exactly one edge on the path between 1 and 3.



How does a cactus look like?

A cactus can be formed by taking a tree, and replacing some subset of its edges with cycles. This is not a complete characterization, but should be enough to build intuition.

Thus, to disconnect 1 and 3 in a cactus, we either cut a bridge, or remove two edges from the same cycle! This gives us $O(M^2)$ candidate edge pairs.



$O(M^3)$

Given the previous idea, we can try removing *every* pair of edges (if we try *all* pairs, we don't need to find the cycles, which is easier to code), and then checking if 1 and 3 are disconnected, and 1 and 2 are connected in $O(M)$. This can be implemented in ~30 lines of code with union-find. Don't forget to also try removing only edge.

$O(M^2 \log(M)^2)$

We can optimize the previous solution mechanically. First, let's check if one edge suffices: try removing every single edge (together with its partner) to handle bridges.

To check if two suffice, first loop over the first edge to remove, and fully remove it and its linked partner. Now, we want to try all other edge candidates in $O(M \log(M)^2)$. This problem can be solved using dynamic connectivity: we can imagine adding all edges, and we then get queries where pairs of edges are removed, and we need to decide whether 1 and 2 are connected, and 1 and 3 are disconnected. We can do this offline using the following trick: https://cp-algorithms.com/data_structures/deleting_in_log_n.html.

You can also get this subtask by writing a slow version of the full solution.

$O(M^2 \log(M)^2)$: simpler using all-but-one trick

There is also a simpler and faster solution. First, let's use the union-find from the link above: it supports adding edges, and removing the last added edge. Then, let's do the following divide and conquer:

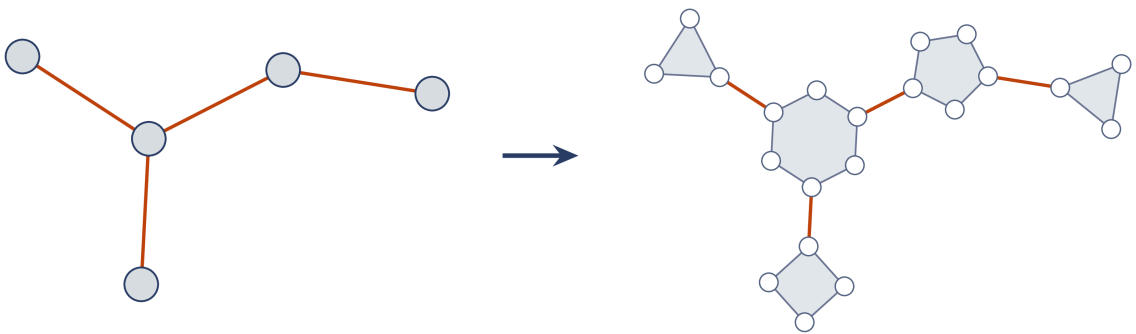
```
def dc(l,r,UF): # invariant: edges [0,l) and (r,m) have been added
  if l==r: # all edges except edge l are in the union-find: same as removing edge l
    # check whether removing l works
  mid=(l+r)//2
  for i in range(l,mid+1): UF.add(edges[i])
  dc(mid+1,r,UF)
  for i in range(l,mid+1): UF.remove_last()

  for i in range(mid+1,r+1): UF.add(edges[i])
  dc(l,mid,UF)
  for i in range(mid+1,r+1): UF.remove_last()
```

We will have to reorder indices of edges so that we only try removing one edge per linked pair, and also that adding each edge adds its partner. Using the master theorem, this runs in $O(M \log(M)^2)$. Blog

Note on subtask “each room is part of at most one cycle.”

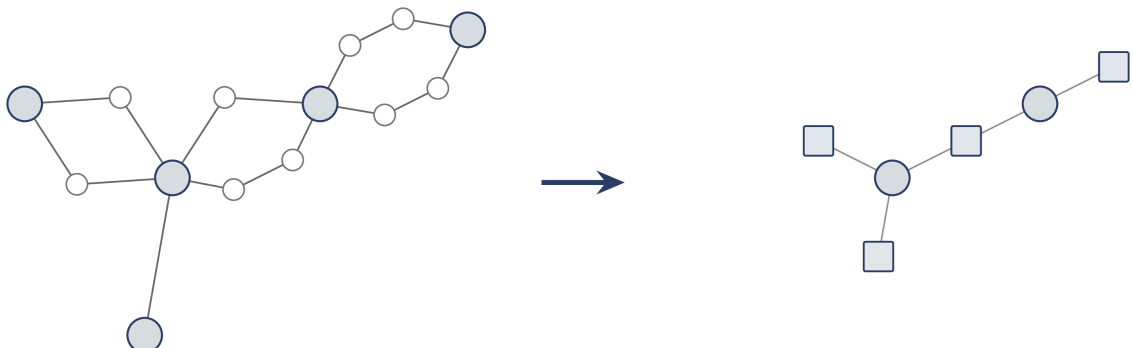
In this subtask, the point is mostly to make it easier to implement. The tree has a simpler shape: instead of replacing some subset of edges with cycles, some vertices are replaced with cycles. This makes it easier to find the cycles: every original tree edge will be a bridge in the expanded graph.

To find the cycles, we can find all bridges and then delete them. Each connected component left is a cycle (potentially of length 1).



Towards a full solution: the block-cut tree

Earlier, we described a cactus as taking a tree and expanding cycles. The opposite can also be done: take a cactus and compress it into its tree structure. This tree is known as the block-cut tree of the graph. We will use this structure to reason about our graph.



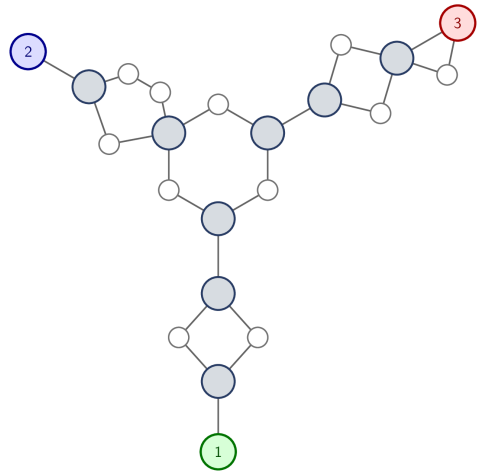
Towards a full solution: the block-cut tree (ii)

Finding the cycles

In general, find the biconnected components and extract the cycles and bridges. There are also cheesier ways: only find the bridges, and then use any spanning tree cleverly. Every non-tree edge closes exactly one cycle, and the cycle is the tree path between its endpoints plus the edge itself.

Towards a full solution: checking if a single edge is enough

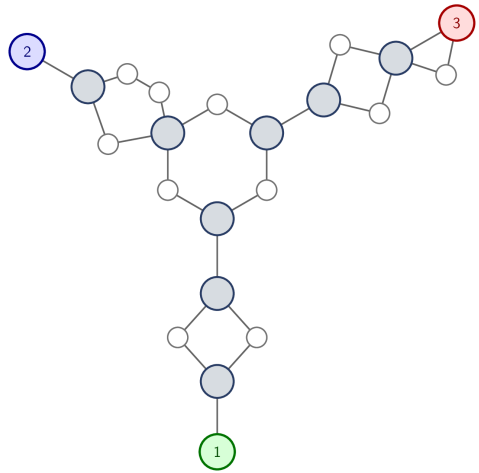
First, notice that the edges and nodes that are “relevant” will form a Y-shape: some shared part up until a cycle where 2 and 3 go into different parts.



Towards a full solution: checking if a single edge is enough (ii)

First, notice that the edges and nodes that are “relevant” will form a Y-shape: some shared part up until a cycle where 2 and 3 go into different parts.

Let O denote the cycle in the middle. Notice that we can never remove a bridge between 1 and O (or two parts of a cycle either).

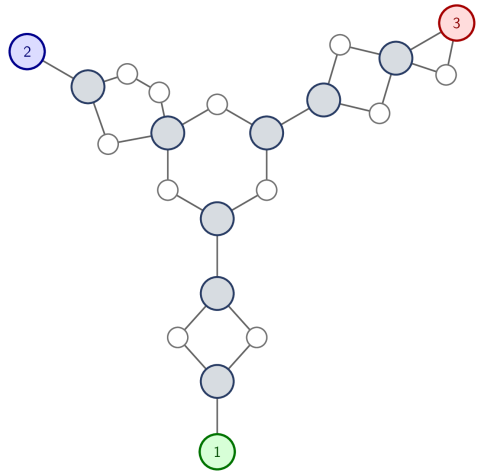


Towards a full solution: checking if a single edge is enough (iii)

First, notice that the edges and nodes that are “relevant” will form a Y-shape: some shared part up until a cycle where 2 and 3 go into different parts.

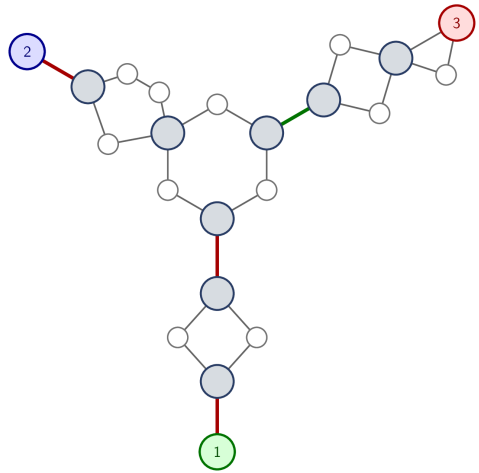
Let O denote the cycle in the middle. Notice that we can never remove a bridge between 1 and O (or two parts of a cycle either).

Thus, we want to remove a bridge between O and 3 that does not lie on the path between 1 and 2.



Towards a full solution: checking if a single edge is enough (iv)

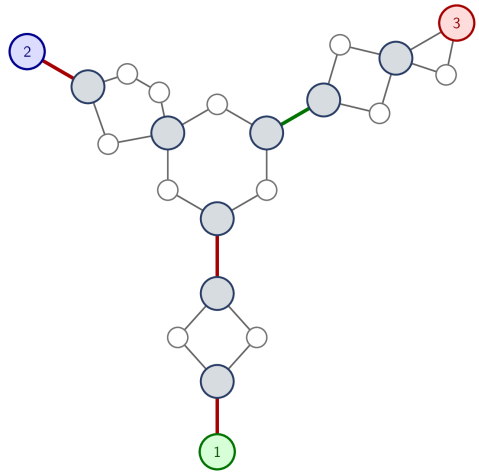
To check this efficiently, first find all bridges. Then take any spanning tree, and mark down the path from 1 to 2 and 1 to 3. For each bridge on each path, mark it either as “necessary” (drawn as red) if on the path 1 to 2, and “potentially cuttable” (drawn as green) if on 1 to 3.



Towards a full solution: checking if a single edge is enough (v)

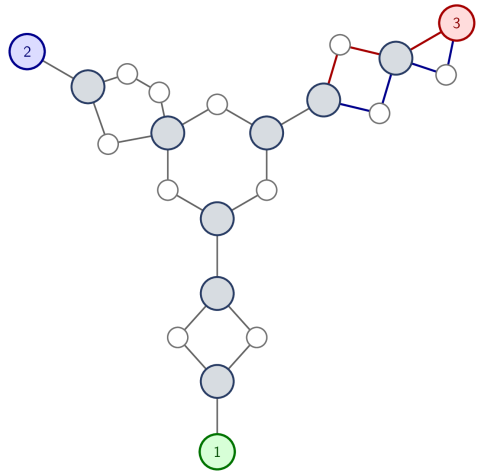
To check this efficiently, first find all bridges. Then take any spanning tree, and mark down the path from 1 to 2 and 1 to 3. For each bridge on each path, mark it either as “necessary” (drawn as red) if on the path 1 to 2, and “potentially cuttable” (drawn as green) if on 1 to 3.

A potentially cuttable bridge wins us the game if its partner is not a necessary bridge. That is the only way it could disconnect us. This condition can be checked in $O(M)$ time.



Full solution: checking pairs of edges (cycle between O and 3)

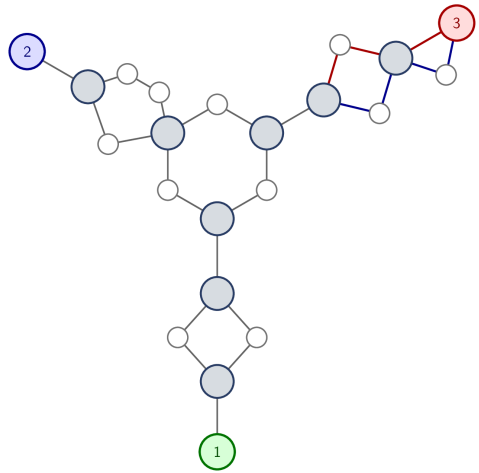
If one edge is not enough, we must cut a cycle. This cycle must not lie between 1 and O, since cutting it would disconnect 1 and 2. Thus, we must cut either O (O needs special care), or a cycle between O and 3.



Full solution: checking pairs of edges (cycle between O and 3) (ii)

If one edge is not enough, we must cut a cycle. This cycle must not lie between 1 and O, since cutting it would disconnect 1 and 2. Thus, we must cut either O (O needs special care), or a cycle between O and 3.

First, let's consider cutting a cycle between O and 3. Notice that we must cut one edge from each "side" of the cycle. To assign sides to cycles, once again take a spanning tree. Every edge used in the path O to 3 can be denoted as being on the "left" side, and the other edges as being on the right side. The left sides from O to 3 are marked as blue, and the right as red. Do the same for the cycles between 1 and 2 for later.



Full solution: checking pairs of edges (cycle between O and 3) (iii)

Finding the cut quickly

Handle each cycle between O and 3 independently. For each cycle, first remove all whose partner cuts a necessary bridge. Then, try looping over each e edge in the left side. Some cases:

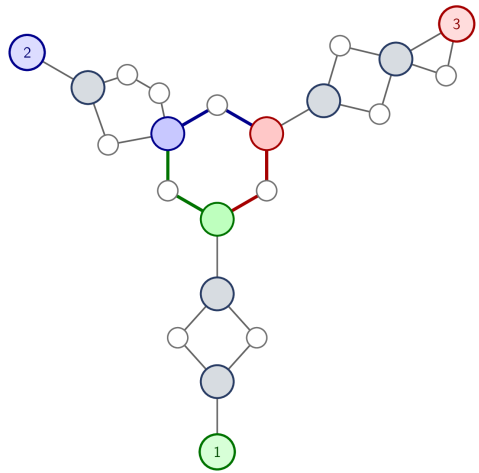
- If e's partner cuts an edge on the right side of the same cycle, the answer is YES and we are done
- If there is any right side edge whose partner cuts a different cycle than e's partner, we can safely take it: cutting two different cycles can never disconnect 1 and 2, so YES and done
- Otherwise, every right side edge cuts the same cycle. We only need to consider at most two such right side edges: one cutting the left side of the other cycle, and one the right side

These cases are exhaustive, and can all be checked in linear time total if implemented carefully.

It might also be easier to view it as a combinatorics problem: count the number of *bad* pairs (left, right) using inclusion-exclusion, and check whether all pairs are bad.

Full solution: cutting O

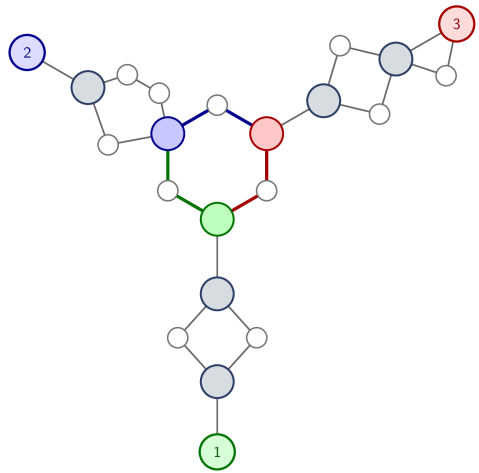
Let's handle O carefully. Note that 1, 2, 3 might lie on O themselves. Still, for each of 1, 2 and 3 there is a vertex on O whose removal removes the vertex itself (e.g. 1) or disconnects it from the rest of the graph; otherwise the graph would not be a cactus. Mark these 3 special nodes: they partition the edges of O into 3 parts, drawn as green, blue and red.



Full solution: cutting O (ii)

Let's handle O carefully. Note that 1, 2, 3 might lie on O themselves. Still, for each of 1, 2 and 3 there is a vertex on O whose removal removes the vertex itself (e.g. 1) or disconnects it from the rest of the graph; otherwise the graph would not be a cactus. Mark these 3 special nodes: they partition the edges of O into 3 parts, drawn as green, blue and red.

The green part must not be cut, directly or indirectly: treat these edges as necessary bridges. Then reuse the cycle solution on the rest, with blue as the "left" side and red as the "right" side.

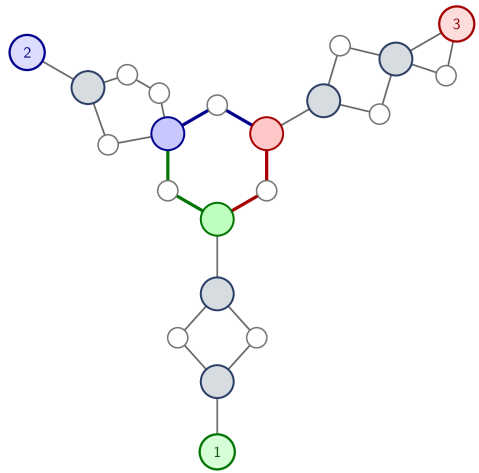


Full solution: cutting O (iii)

Let's handle O carefully. Note that 1, 2, 3 might lie on O themselves. Still, for each of 1, 2 and 3 there is a vertex on O whose removal removes the vertex itself (e.g. 1) or disconnects it from the rest of the graph; otherwise the graph would not be a cactus. Mark these 3 special nodes: they partition the edges of O into 3 parts, drawn as green, blue and red.

The green part must not be cut, directly or indirectly: treat these edges as necessary bridges. Then reuse the cycle solution on the rest, with blue as the "left" side and red as the "right" side.

Don't forget the degenerate cases where some color has no edges at all, or when O is a single node.



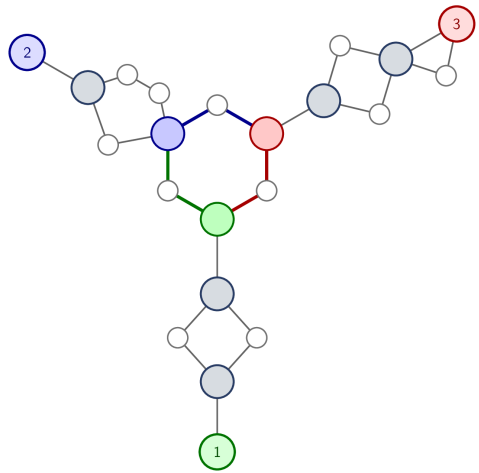
Full solution: cutting O (iv)

Let's handle O carefully. Note that 1, 2, 3 might lie on O themselves. Still, for each of 1, 2 and 3 there is a vertex on O whose removal removes the vertex itself (e.g. 1) or disconnects it from the rest of the graph; otherwise the graph would not be a cactus. Mark these 3 special nodes: they partition the edges of O into 3 parts, drawn as green, blue and red.

The green part must not be cut, directly or indirectly: treat these edges as necessary bridges. Then reuse the cycle solution on the rest, with blue as the "left" side and red as the "right" side.

Don't forget the degenerate cases where some color has no edges at all, or when O is a single node.

It is probably possible to handle O without special-casing it.



Full solution: cutting O (v)

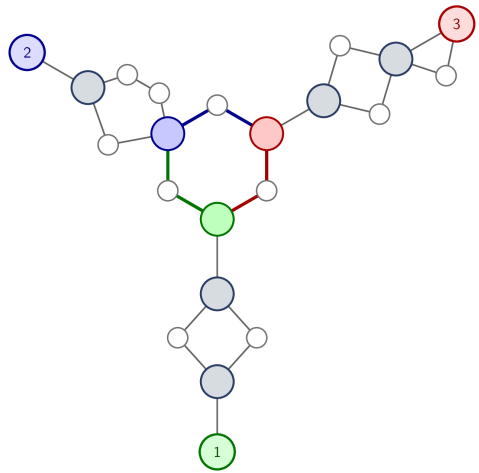
Let's handle O carefully. Note that 1, 2, 3 might lie on O themselves. Still, for each of 1, 2 and 3 there is a vertex on O whose removal removes the vertex itself (e.g. 1) or disconnects it from the rest of the graph; otherwise the graph would not be a cactus. Mark these 3 special nodes: they partition the edges of O into 3 parts, drawn as green, blue and red.

The green part must not be cut, directly or indirectly: treat these edges as necessary bridges. Then reuse the cycle solution on the rest, with blue as the "left" side and red as the "right" side.

Don't forget the degenerate cases where some color has no edges at all, or when O is a single node.

It is probably possible to handle O without special-casing it.

It is possible to implement the entire solution in $O(M)$ time.



2 Lieutenant's Lotto

Lieutenant's Lotto

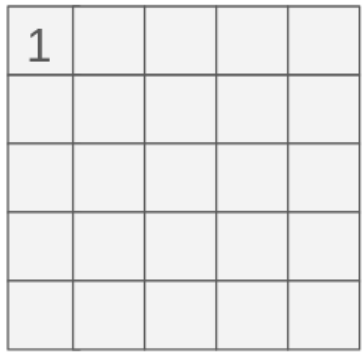
Author: Nils Gustafsson

Description

You are given a permutation $p_1, p_2, \ldots, p_N$. Construct an $N \times N$ binary grid where, for every i, reading row i left to right gives the same string as reading column p_i top to bottom, and all N rows are pairwise distinct. If no such grid exists, report that it is impossible.

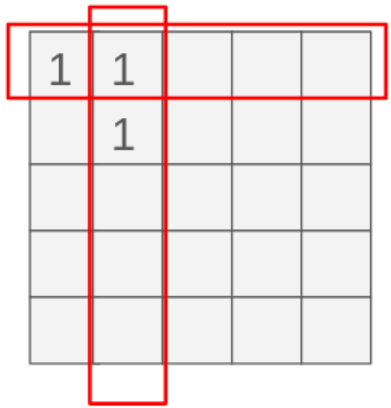
Lieutenant's Lotto

Subtask 3: $p_i = i + 1$



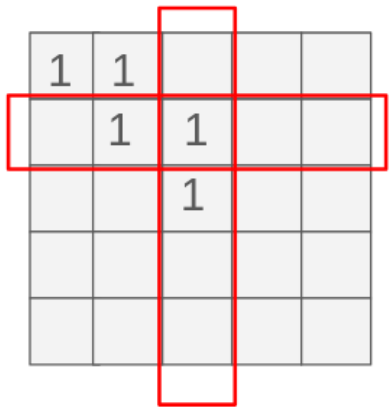
Lieutenant's Lotto

Subtask 3: $p_i = i + 1$



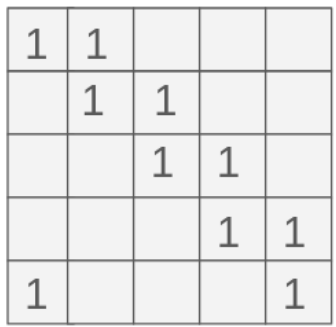
Lieutenant's Lotto

Subtask 3: $p_i = i + 1$



Lieutenant's Lotto

Subtask 3: $p_i = i + 1$

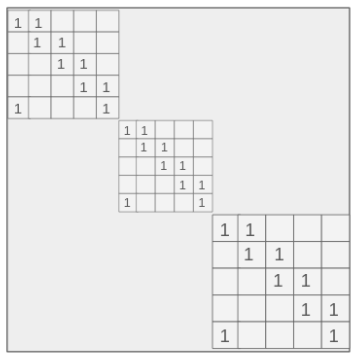


Put zeros everywhere else, and you're done. $N = 2$ is impossible.

Lieutenant's Lotto

Subtask 4: $p_{p_i} \neq i$

Decompose the permutation into cycles. The solution from Subtask 3 can be generalized to any cycle: Put a 1 on $(i, i), (i, p_i), (p_i, p_i), \left(p_i, p_{p_i}\right), \ldots$ and so on. Since there are no cycles of length 2, this always works.

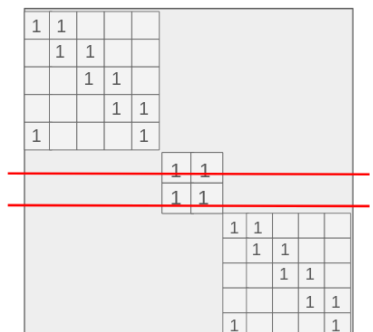


The cycles are not always nice and adjacent like in the picture, but that turns out to not matter.

Lieutenant's Lotto

Full solution

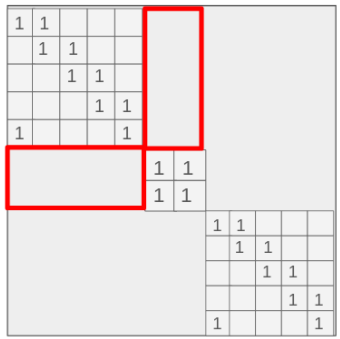
We want to use the idea from Subtask 4, but cycles of length 2 are ruining it. Their rows become equal...



Lieutenant's Lotto

Full solution

We need some other cycle to help them make their rows different, by putting different numbers in the intersection (cells (i, j) where one of i, j belongs to the 2-cycle and one belongs to the other cycle).



Lieutenant's Lotto

Full solution

Let's think of the problem as a graph coloring problem. We add edges between (i, j) and (j, p_i). We want to color the connected components such that all rows are different.

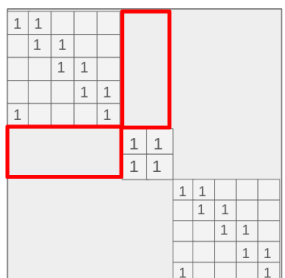
Note: For a cell (i, j), consider the cycles that i and j belong to in the permutation. In the graph we built, every cell will belong to these two cycles.

In our case, this means that the intersection region of the grid that we looked at on the previous slide is disconnected from the rest of the grid (in the graph).

Lieutenant's Lotto

Full solution

For every 2-cycle, try every other cycle and see if the intersection region consists of more than one connected component in the graph. If yes, color them differently and we are done. Otherwise it is impossible. It turns out that this works if and only if there is at least one more even cycle.



Lieutenant's Lotto

Cheese solution

Build the graph with edges between (i, j) and (j, p_i). Color the components randomly, and check if all rows become different. If not, repeat a couple of times.

It is probably impossible to make test data that reliably stops this, but we tried to make it annoying to solve the problem this way.

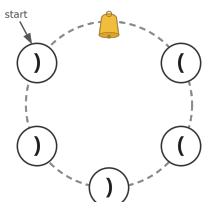
3 The Cat Necklace

The Cat Necklace

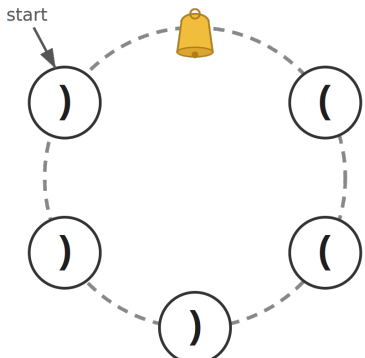
Authors: Harry Zhang, Nils Gustafsson

Description

Given a string of length N consisting of "(" and ")". You can remove any character for a cost of a, and put the first character at the end for a cost of b. Calculate the minimum cost to make the string balanced.

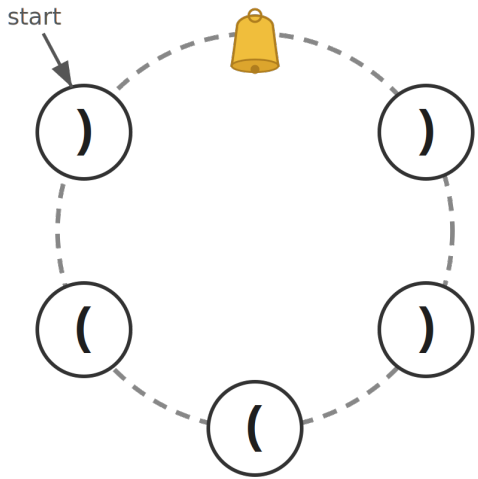


The Cat Necklace: Sample 1 solution



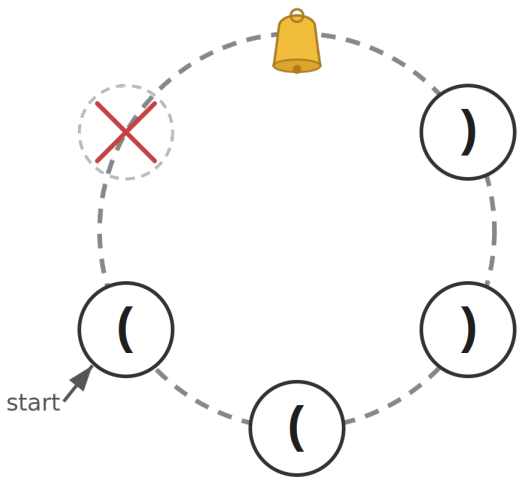
After rotating it clockwise twice

)(())



If we remove the first charm, it is balanced

`(()) – balanced!`



The Cat Necklace ($a = 1, b = 10^9$)

The answer is at most $N \cdot a = N \leq 5 \cdot 10^5$, since we can remove all brackets. Thus, we never want to rotate the string.

This subtask reduces to wanting to remove the minimum number of brackets, such that the string becomes balanced, without rotating.

Can be solved using the standard stack-solution. Iterate from left to right, maintain a stack of all unmatched left-parenthesis, and for each right-parenthesis, match it with the most recent left-parenthesis.

The Cat Necklace ($a = 10^9, b = 1$ and same number of left as right)

In this case, we want to remove as few brackets as possible, meanwhile rotating the string is basically free.

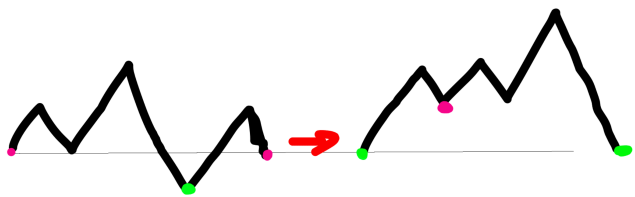
Question

Given a string of length N consisting of "(" and ")", where we have the same number of "(" as ")". What is the minimum number of brackets to remove?

The Cat Necklace ($a = 10^9, b = 1$ and same number of left as right) (ii)

Answer: 0

Proof:



Since same number of left and right, the level-graph will start and end at $y = 0$.

Assume there is some point that is under the line $y = 0$. Pick the earliest lowest point, and rotate the string so it starts at that point. Then all positions of the level-graph will be above $y = 0$.

The Cat Necklace ($b = 0$)

With the same reasoning, this subtask has this following simple solution:

```
n, a, b = map(int, input().split())
s = input()
l = s.count('(')
h = s.count(')')
print(abs(l - h) * a)
```

The Cat Necklace ($N \leq ?$)

Moving on to the main problem.

Let's try some simple polynomial solutions.

What if we try all possible N starting positions in the string, and for each rotation we could just run the stack-solution from subtask 1, leading to a $O(N)$ solution?

That should work... right?

The Cat Necklace ($N \leq ?$) (ii)

Nope!

Consider the case:

```
7 4 1
()))(((
```

By running the previously described solution, we are sometimes considering rotating the string first, and then removing the rotated brackets.

But in this case, we actually want to remove the first bracket, and then rotate the string 3 times.

The Cat Necklace ($N \leq 15$)

This gives us the core idea of the solution. In the optimal solution, we always want to remove brackets before rotating anything at all.

A correct bruteforce solution would then to consider all 2^N subsets of brackets to remove. In the reduced string where we have removed the given subset of brackets, we try all $O(N)$ rotations, and for each of them run the stack-solution from before.

This runs in $O(2^N N^2)$.

The Cat Necklace ($N \leq 2000$)

Suppose we decide on a rotation offset c (we want to read starting from position c). The rotated string is $s[c..n-1] + s[0..c-1]$.

We need to choose which characters to keep (forming a balanced subsequence). The total cost has two parts:

- a per removed character.
- Each character from $s[0..c-1]$ that we keep forces one rotation step (cost b). But each character from $s[0..c-1]$ that we remove before rotating saves one rotation step.

The Cat Necklace ($N \leq 2000$) (ii)

So the cost is:

$$\text{cost}(c) = a \cdot d + b \cdot k$$

where d = total removals and k = characters kept from $s[0..c-1]$.

Equivalently, letting $S = c - k$ (removals from $s[0..c-1]$):

$$\text{cost}(c) = b \cdot c + a \cdot R - b \cdot S$$

where R = minimum removals for the rotated string, and S = maximum number of those removals that come from positions $0..c-1$.

The Cat Necklace ($N \leq 2000$) (iii)

Define prefix sums $P[i]$ where `(` contributes $+1$ and `)` contributes -1. Let $\text{tot} = P[n]$.

For rotation c, the prefix sums of the rotated string have minimum:

$$M - P[c], \quad \text{where } M = \min(\text{mr}[c], \text{tot} + \text{ml}[c])$$

with $\text{mr}[c] = \min(P[c], ..., P[n])$ and $\text{ml}[c] = \min(P[0], ..., P[c])$.

The minimum removals are:

$$R = \underbrace{(P[c] - M)}_{\text{unmatched)}} + \underbrace{(\text{tot} + P[c] - M)}_{\text{unmatched (}} = 2P[c] + \text{tot} - 2M$$

The Cat Necklace ($N \leq 2000$) (iv)

For a fixed c, we compute S by processing the rotated string with a greedy algorithm that maximizes removals from $s[0..c-1]$.

Maintain two counters: cnt_1 for unmatched `(` from part 1 ($s[c..n-1]$) and cnt_2 for unmatched `(` from part 2 ($s[0..c-1]$). Process left to right:

- On `(`: increment cnt_1 or cnt_2 depending on which part the character belongs to.
- On `)`: if $\text{cnt}_1 > 0$, match with a part 1 `(` (decrement cnt_1). Otherwise if $\text{cnt}_2 > 0$, match with a part 2 `(` (decrement cnt_2). Otherwise, this `)` is unmatched.

The key: always prefer matching with part 1. This leaves part 2 `(` unmatched, which counts toward S.

After processing: S = (unmatched `)` from part 2) + cnt_2.

This is $O(N)$ per rotation, giving $O(N^2)$ total.

The Cat Necklace (Full Solution)

To go from $O(N^2)$ to $O(N)$, we need to compute $S(c)$ in $O(1)$ instead of $O(N)$.

Key claim: no `)` from part 2 is unmatched

Fix a rotation offset c with $\mathrm{mr}[c] \leq \mathrm{tot} + \mathrm{ml}[c]$ (the prefix-sum minimum is in part 1).

After processing part 1 ($s[c..n-1]$) with the greedy, the number of unmatched part 1 `(` remaining is:

$$\mathrm{cnt}_1 = \mathrm{tot} - \mathrm{mr}[c]$$

(The first part creates $P[c] - \mathrm{mr}[c]$ unmatched `)` and ends with $\mathrm{tot} - P[c]$ net `(`, giving $\mathrm{cnt}_1 = (\mathrm{tot} - P[c]) + (P[c] - \mathrm{mr}[c])$.)

During part 2 processing, a `)` is unmatched only when both $\mathrm{cnt}_1 = 0$ and $\mathrm{cnt}_2 = 0$. But the total available `(` at position j in part 2 is always $\geq \mathrm{cnt}_1^{\mathrm{init}} + P[j] \geq (\mathrm{tot} - \mathrm{mr}[c]) + \mathrm{ml}[c] \geq 0$ (by our condition). So no part 2 `)` is ever unmatched, and S consists entirely of unmatched part 2 `(`.

The Cat Necklace (Full Solution) (ii)

Let $L_2 = \frac{c+P[c]}{2}$ be the number of `(` in part 2, and $R_2 = \frac{c-P[c]}{2}$ the number of `)`.

Case 1: $\mathrm{cnt}_1 \geq R_2$. Then cnt_1 never runs out. Every `)` in part 2 matches with a part 1 `(`, so no part 2 `(` is consumed. All L_2 are unmatched: $S = L_2$.

Case 2: $\mathrm{cnt}_1 < R_2$. Then cnt_1 is fully consumed. The remaining $R_2 - \mathrm{cnt}_1$ closing brackets match with part 2 `(`. The unmatched part 2 `(` are $L_2 - (R_2 - \mathrm{cnt}_1) = P[c] + \mathrm{cnt}_1 = \mathrm{tot} + P[c] - \mathrm{mr}[c]$. This equals the total number of unmatched `(` in the entire string, so all of them come from part 2.

Both cases combine into:

$$S = \min(L_2, \mathrm{tot} + P[c] - M) = \min\left(\frac{c + P[c]}{2}, \mathrm{tot} + P[c] - M\right)$$

The Cat Necklace (Full Solution) (iii)

When $\mathrm{mr}[c] > \mathrm{tot} + \mathrm{ml}[c]$ (the prefix-sum minimum is in part 2), the formula above becomes a safe underestimate: it correctly counts unmatched `(` from part 2, but misses unmatched `)` from part 2 that also contribute to S. However, the global minimum over all c is preserved, since whenever this underestimates at rotation c, a smaller c' achieves equal or better cost.

Precompute P, ml and mr in $O(N)$. For each c, compute R, M, and S in $O(1)$. Total: $O(N)$ time and space.

There are other ways to solve it in $O(N \log N)$ time, as well as other linear time solutions.