Problem A. Apple Tree

It can be shown that the depth of the apple tree is $O(\sqrt{n})$. All paths of the tree are no longer than twice the depth, so after any re-rooting, the depth is $O(\sqrt{n})$.

We will run a DFS that does a re-rooting along the way, and also maintain $cnt_{v,d}$: the amount of vertices d edges away from v. Then having a center v determined, we add $\sum \binom{cnt_{v,d}}{k-1}$ to the answer.

One important consideration in this problem is the amount of memory used. The size n is rather large for a typical sqrt-based problem, and that's because a lot of vertices have very small subtrees. We can say that a vertex has depth d only if it has at least d children. So when we create original depth counters, the total amount of memory used is $\sum d_v \le \sum deg_v = O(n)$. This observation doesn't affect time complexity, as after re-rooting, the apple tree property is destroyed, but all entries to counters are just moved entries from the original linear time.

The time complexity is $O(n\sqrt{n})$, and the space complexity is $O(n)$.

Problem B. Balatro

We need some sort of a knapsack for the problem, but we can't do a classic one as the weights are too high. Fortunately, we can split the cards into two groups, where each group has small values in one of the dimensions.

Now we solve knapsacks independently for both sets, finding optimal $dp_{k,\sum a} = \sum b$. Having this dp and symmetric dp', we can perform the merging:

$$ans = \max_{i=0}^{k} \max_{w=0}^{W} \max_{w'=0}^{W} (dp_{i,w} + w') \cdot (dp'_{i,w'} + w).$$

Problem C. Classement Nationale

Well, you just need to iterate over all subsets and then simulate the process. Some tricks to consider:

1. As the computation of CN depends on 15 days before it, the computation is invariant to race permutation as long as days are non-decreasing: so they can be processed just as given in input.
2. Data is sparse: it means each simulation should take not $n \cdot m$ but $\sum k_j$ operations.
3. It makes sense to use three pointers: one for the current competition, another for pending updates from competitions 15 days before it, and one more to clean up old records of 365 days ago.

Problem D. Digit Division

Preliminaries
First, let's decompose $k = 2^a \cdot 2^b \cdot k'$, where k' is coprime with 10. If we solve the problem for k', then we can add $max(a, b)$ zeros to the end of the answer: the sum of digits doesn't change and the number becomes divisible by $10^{max(a,b)} \implies$ also by k.

The answer exists for any k', we can always find such x that $10^x \equiv 1 \pmod{k}$, so correct n could be $\sum_{i=1}^{i=k} 10^{ix}$. The sum of digits is k and the number is divisible by k (as the sum of k terms that are $\equiv 1 \pmod{k}$).
This approach gives an answer of the length k^2 (or $\frac{k^2}{9}$), while we are required to use at most $20k$ digits.

For convenience, let's now use k instead of k'
Main idea
First idea is that we can concatenate number k to itself multiple times, getting $\overline{kk\ldots kk}$. This can be

written as $\sum 10^{\alpha_i} k$, so it's obviously divisible by k.

If k is divisible by sum_digits(k), the problem is solved. Sometimes it's not, and that's what we have to solve.

Original idea
We can use not only k, but also $a_i k$ for different a_i.
Once we came up with a *good* set of a_i, we can just use knapsack to get $\sum\limits_{i_1, i_2, \ldots i_p} \text{sum\_digits}(a_i k) = k$.

A sufficient set of a_i was $[1, 3, 5, 11, 33, 55, 111, \ldots, 555555]$.
It just works, besides several small cases that can be bruteforced.
Note: we should be careful with picking a_i, for example if $k = 10\,001$ then $\forall a_i \le 9999 : \ a_i k = \overline{a_i a_i} \implies \text{sum\_digits}(a_i k) = 2\,\text{sum\_digits}(a_i) \implies$ knapsack is unsolvable since we want to reach an odd number $k = 10\,001$ using only even numbers.

Alternative approach 1
We can find such x that $[1, x]$ is sufficient.
We need such x that $\text{sum\_digits}(xk) \equiv k \pmod{\text{sum\_digits}(k)}$.
We can just run a search starting from $x = 1$. It will find such x quite fast since $\text{sum\_digits}(k) < 50$ and sum_digits(xk) is quite random.

Alternative approach 2
Let's generate some numbers $n_1, n_2, \ldots, n_p$ ($p \approx \sqrt{k}$) such that $\text{sum\_digits}(n_i) = \frac{k}{2}$.
We need to find n_i and n_j such that $n_i + n_j \equiv 0 \pmod{k}$.
Again, we will likely find such n_i and n_j thanks to the birthday paradox.
The problem here is that we cannot (or don't want to) generate $\sqrt{k}$ numbers of length k.
Instead, we can start with n_1, then do some modifications, one digit by one digit, keeping a persistent trace of the numbers that we have until we find required n_i and n_j.

Alternative approach 3
Let's generate a number of length $2k$ with k zeros and k ones.
Let's do random modifications to it. The results that we get modulo k are quite random, so we should find a correct answer in $O(k)$ steps.

Problem E. Euclid in Manhattan

We can notice that the optimal walking strategy is the alternation of down and right, skipping several blocks while crossing the street (or avenue) Euclid is moving along.

So this formulates two DPs: one for bottom-left corners of buildings, another for top-right corners:

$$dp^{bl}_{i,j} = \min_{k=1}^{j} dp^{tr}_{i-1,k} + \sqrt{w_i^2 + (\sum_{l=k}^{j} h_l + H_l)^2}$$

$$dp^{tr}_{i,j} = \min_{k=1}^{i} dp^{bl}_{k,j-1} + \sqrt{h_i^2 + (\sum_{l=k}^{j} w_l + W_l)^2}$$

The order to calculate these DPs is by enumerating the diagonals. This gives an $O(n^3)$ solution.

To optimize the DP, we need some observations on the structure of the answer. Let's consider traversing diagonally some street as a part of (so far) some optimal trajectory, starting in coordinates $A < B$ and arriving in C, D respectively. We can notice that AC should not intersect with BD. Otherwise:

$$\begin{cases} dp_A + AC < dp_B + BC \\ dp_B + BD < dp_A + AD \end{cases}$$

$$AC + BD < BC + AD$$

And the line above is not true for crossed lines.

So we know that having $i < i'$ for a fixed j, optimal segments that follow (i, j) and (i', j) do not intersect, and the symmetric stands as well.

For each avenue/street independently, we'll store a structure that shows what is the segment of indices that this currently known dp's are optimal for. They form segments of coordinates, and new added segment is optimal at infinity, so it pops several previous elements from the stack. We need to be able to find a point where one dp becomes better than the other, which means $dp_1 + \sqrt{w^2 + (x - x_1)^2} = dp_2 + \sqrt{w^2 + (x - x_2)^2}$. This can be solved analytically, but binary search is also an option that should pass.

So, for each street or avenue we now keep this data structure that gives the best matching corner to any coordinate. We'll need to append to this structure and do the search. As we have the specific order of calculating the DP, we'll manage all these data structures in parallel.

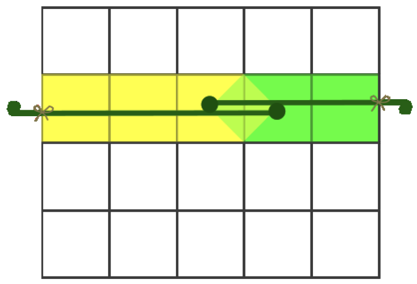
The model solution has $O(nm \log n)$ complexity. However, it can be sped up to $O(nm)$.

Problem F. Framboise

Preliminaries

- Each segment of the fence can take at most 1 stem by construction, since the cell next to this segment can be covered only once.
- For each segment, we can choose the closest stem that can reach that segment; it just occupies a smaller area than any other option. Let's remove all other stems.
- Now, two stems can block each other only if one of them is vertical and the other is horizontal.

To prove the last point, we need to show that it's impossible for two horizontal stems to block each other.



First, they need to be in the same row. Second, they need to go in opposite directions. This is impossible: Let's say that the right one of them is longer. If they intersect, the left looks right, the right looks left. But since the right one is longer and closer to the right side of the fence, we would have chosen this stem as the closest.
Contradiction.

Solution
Now we know that if two stems block each other, they are perpendicular.
Let's build a bipartite graph where the left part contains all horizontal stems and the right part contains all vertical ones.
Let's create an edge between two vertices if they block each other.
We need to find the maximum independent set in this graph.
It's a well-known task; we need to build a maximum matching(Kuhn/Dinic) first, and then retrieve the corresponding set.
(Google "Maximum Independent Set in Bipartite Graphs" and "Kőnig's theorem" for more details on this).

Problem G. Goofy Songs

In the problem, we are asked to match the two-line blocks against a pattern and then find the maximal segment that has a repeating pattern.

We can do all of this ourselves, but a nicer way would be to use regexp's for that.

There is a rather complex regular expression for the whole problem, but basic grouping can be used to parse a single line as well. With groups it is easier to find block generator $\mathbb{S}$ and keep current best score.

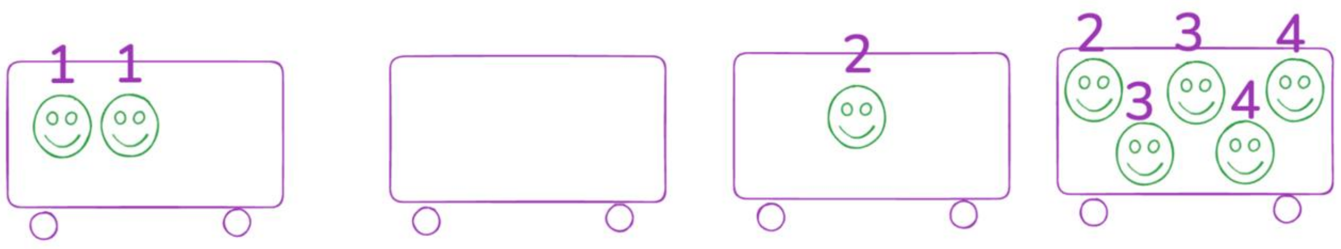
```
r’([a-z]+), \1 \1ity \1 \n’ r’i said \1, \1 \1ity \1\n’
```

The full regexp solution requires some look-before regexp checks, that allow regexp to check the previous symbols without overlapping with another matches:

```
r’(?<![^\n])([a-z]+), \1 \1ity \1\ni said \1, \1 \1ity \1\n (\1, \1 \1ity \1\ni said
\1, \1 \1ity \1\n)*’
```

Problem H. Heure de Rush

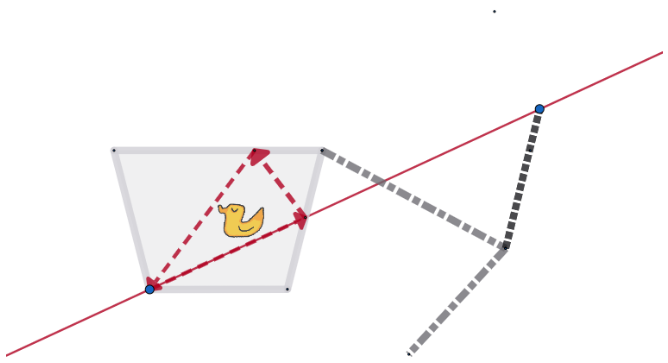
Preliminaries
Let's number all people from left to right (arbitrarily inside one car).
If someone has a smaller number than someone else, he will end up also to the left. Otherwise, we could just swap them at the moment of overtaking.
For every person, we know where they will end up.



Solution
For every car, all people would end up in some segment of cars: from l_i to r_i. We care only about the leftmost and the rightmost.
So we just compute l_i and r_i. The answer is the maximum time to reach them over all cars.

Problem I. Infrared

The key observation is the classical unfolding/mirroring trick: instead of bending a ray at each reflection, we reflect the polygon across the wall, so that in the unfolded plane the trajectory becomes a single straight segment. After reflecting across walls $2, \ldots, n-1$ (in order), the original vertex p_1 has an image q_n. A valid laser path exists iff the straight ray q_0q_n (where $q_0 = p_1$) intersects every unfolded edge $[q_i, q_{i+1}]$ strictly inside. So the solution is to unfold the polygon into a segment a verify the ray found.



Unfolding.

We'll unfold the polygon sides one by one. To do that, we think of each side as of (rotation, length) pair relative to the previous side. We just need to stack them together, additionally mirroring the rotation on every even side.

Suppose in the original polygon we know p_{i-2}, p_{i-1} and in the unfolded plane their images q_{i-2}, q_{i-1}. We want to recover the rotation, as the length doesn't change. Let

$$\text{old} = p_{i-1} - p_{i-2}, \quad \text{dir} = q_{i-1} - q_{i-2}.$$

We can recover the cos and sin of the needed angle with dot and cross products: $(\frac{old \cdot dir}{|old|^2 \cdot |dir|^2}, \frac{old \times dir}{|old|^2 \cdot |dir|^2})$. And now we can perform a new rotation using given cos and sin.

Intersection test. For each unfolded edge $(a, b) = (q_i, q_{i+1})$ and ray $(c, d) = (q_0, q_n)$, we want relative coordinate of an intersection point w_i:

$$w_i = a + t \cdot (b - a) \quad w_i = c + u \cdot (d - c)$$

We'll explain how to derive t.

$$a+t\cdot(b-a) = c+u\cdot(d-c) \quad (a-c)\times(d-c)+t\cdot((b-a)\times(d-c)) = u\cdot((d-c)\times(d-c)) \quad t = \frac{(c-a)\times(d-c)}{(b-a)\times(d-c)}$$

We require $t \in [\varepsilon, 1-\varepsilon]$ with $\varepsilon = 10^{-5}$ (strictly interior) and $u \geq 0$ (forward along the ray). Failure means "NO". If all pass, the firing direction is simply $q_n - q_0$, so the answer is "YES" and the angle is $\theta = \text{atan2}(d_y, d_x)$ in degrees.

Intended complxtity is $O(n)$. The low limits are due to numerical instability and the fact that it is hard to verify the model solution precisely enough. We tried to have rather forgiving tests and room for variance in computations.

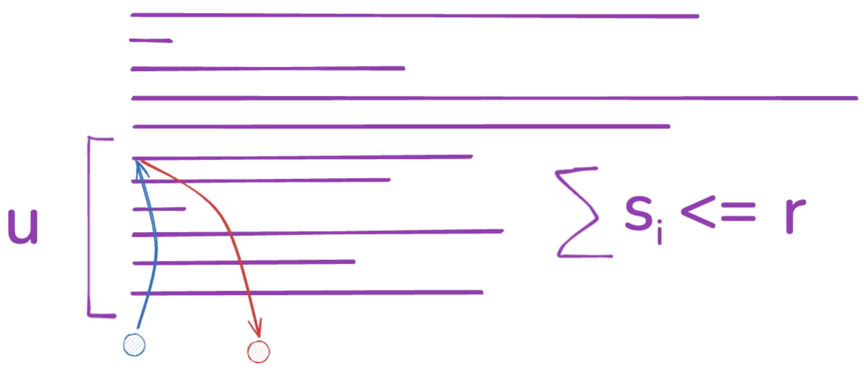
Problem J. JamBrains

Preliminaries

Top U+1 rows are always successful. We win in the first round.
Not that clear for other rows.
For every row, either all or none positions are successful. We'll prove it further.

Solution

First, let's solve the problem without updates.
Let's consider some block of exactly U rows that has the sum of lengths $\sum_i s_i \le R$



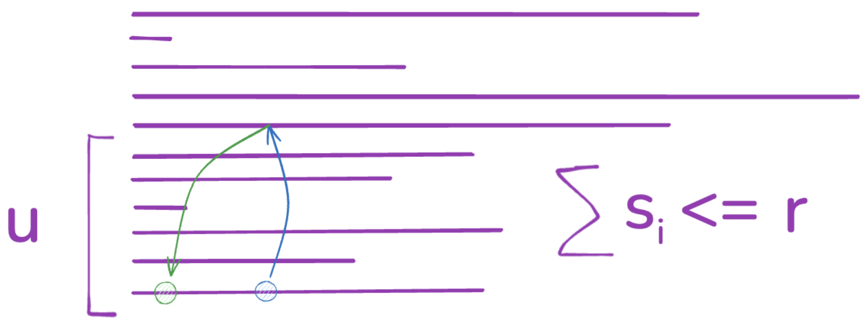
Let's consider a blue point in any line below this U rows.
In one round, doing up clicks(blue arrow), we can reach at most the first symbol of the top line of our U rows.
Easy to see that doing R right clicks(right arrow) we will end up below our U rows again.
This proves that all positions below such a block of U rows are not successful.
Let's drop all lines below the very first such block (if it exists).

From now,

$$\sum_i s_i > R \tag{1}$$

(1) holds for any U consecutive rows.

Let's prove that all remaining positions are successful.
Let's pick any blue point and prove that after every round we move closer to the first row(end up above or to the left).



Let's denote our blue position as (i, j). After U jumps up we end up at $(i - U, \min(j, s_{i-U}))$.
Let's prove that we need **more than** R right clicks to go from $(i - U, \min(j, s_{i-U}))$ to (i, j).

- To reach the end of line $i - U$ we need: $s_{i-U} - \min(j, s_{i-U})$ clicks.

- To descend $U-1$ rows down we need: $\sum\limits_{i=i-U+1}^{i-1} s_i$ clicks.
- To reach (i, j) we need: j clicks.

In total we need $(s_{i-U} - \min(j, s_{i-U})) + \sum\limits_{i=i-U+1}^{i-1} s_i + j$ clicks.

$j - \min(j, s_{i-U}) \geq 0 \implies$
$\implies (s_{i-U} - \min(j, s_{i-U})) + \sum\limits_{i=i-U+1}^{i-1} s_i + j = (j - \min(j, s_{i-U})) + \sum\limits_{i=i-U}^{i-1} s_i \geq \sum\limits_{i=i-U}^{i-1} s_i > R$ by (1).

So the solution is to find first such $p > 1$ that $\sum\limits_{i=p}^{p+U-1} s_i \leq R$ and take $\sum\limits_{i=1}^{p+U-1} s_i$ as the answer.

Adding updates
We need to solve 2 tasks:

- Find the first block of U rows that has a sum of lengths $\leq R$.
- Take the sum of some prefix.

The second task is easy enough; use any tree you like that supports position updates and prefix sums.
The first one is a bit more tricky.
Let's make a segment tree that at position i stores the sum of lengths of U rows starting from i.
And make this segment tree answer prefix minimum queries.
When we get an update, we need to update some segment of values (those that contain us in their block of U rows).
Now, to find the first such block, we can binary search the prefix that has a value $\leq R$.
The complexity is $O(n \log^2 n)$. It can also be done in $O(n \log n)$, but we didn't ask for that; binary searching passes freely.

Binary search approach
Without updates, you can binary search the answer (by rows or even purely by positions) using some simulation of the rounds.
If you know how to adapt this solution to the case with updates, please let us know.

Problem K. k Operations

It can be seen that operations could be applied one by one in a greedy way. And for a single operation, it is optimal to subtract from the minimal value left in the array.

So, basically, k operations decrease some elements to 1, leave all the large elements untouched, and decrease one middle element by the rest. So we need to put elements in sorted order in a way, find the biggest prefix with $\sum(a_i - 1) \leq k$, and then decrease the last element by $k - \sum(a_i - 1)$.

To do this, you can have a persistent segment tree that stores a counter for each value in some prefix of the array. To answer a query on $[\ell, r]$, one can take $t_{\ell-1}$ and t_r, then have a "virtual" tree that has the difference in prefix sums for two trees. After you go down in this tree, you find the middle element, get the product of all bigger elements, and update the middle element.

Everything can be (carefully) implemented in $O((n+q)\log n)$.

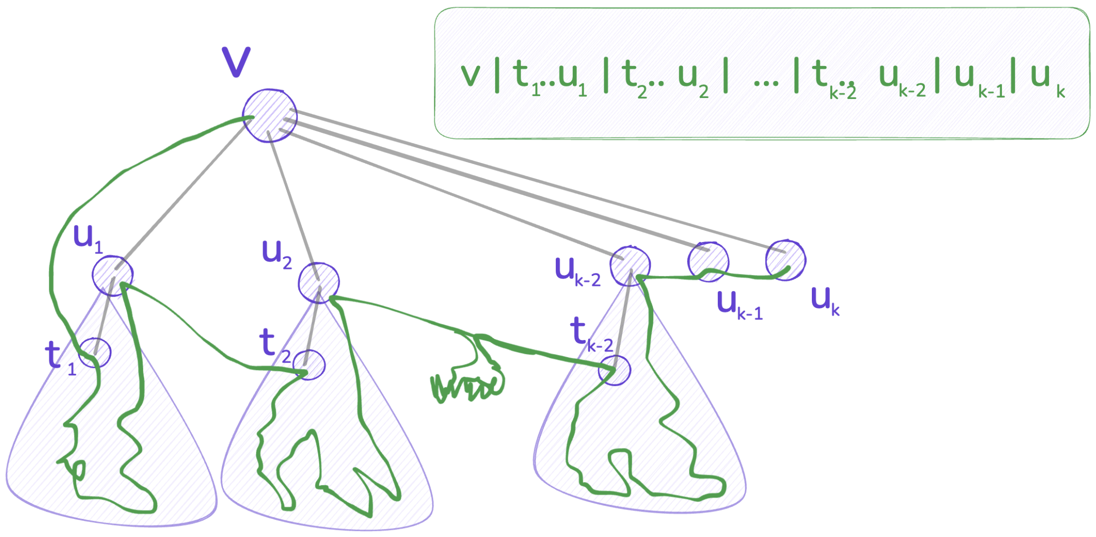
Problem L. Lice Hopping

Preliminaries

- If we can jump at distance 1, the only valid graph is bamboo.
- We can always traverse a tree using jumps of length at most 3.
- So, we have to determine if it's possible with jumps of length at most 2.

Subtask: Proving that answer ≤ 3
If we can jump at distance 3, we can make the tree rooted and by induction solve the following problem: For vertex v, we want to traverse the whole subtree of v starting at v and finishing at some direct child of v (besides leaves, for them we finish also at v).
Base case: leaves.
Induction step: we solved this problem for every child.
Now let's explicitly build a traversal. Let's say we have k direct children.

- We start at v.
- We append the reversed traversal for every one of our children, one by one.



Easy to see that connections between blocks that we have concatenated are valid. We always have a child on one end and a child or a grandchild on the other end of the edge, so the distance is at most 3. The end of the last block is one of our children, so the induction step is finished.

Solution
Lemma0: If we can traverse a tree with jumps of length at most 2, we can also traverse any of it's connected subgraphs.
Proof0:
Let's do it step-by-step removing leaves to converge to the subgraph.
Let's prove that we can remove one leaf.
Let's look at the moment when it appears in the traversal. If it's the end, we won. Otherwise, it has 2 neighbors in the traversal. For a leaf, it can reach it's neighbor (let's call it v), or any of v's neighbors. We can see that any of those points are at distance at most 2, so we can safely remove the leaf.

Let's call a vertex *interesting* if it has at least 3 neighbors that are not leaves.

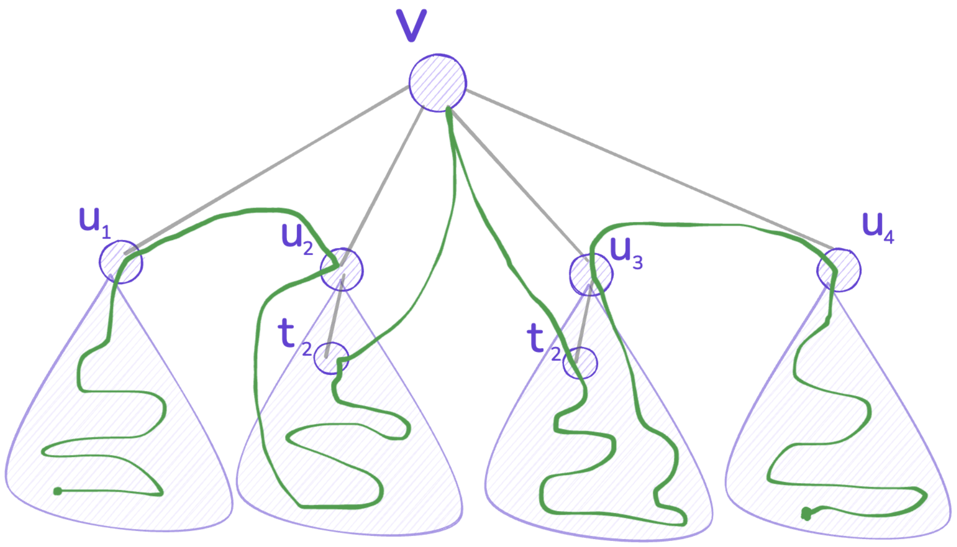
Lemma1: If there is a vertex that has at least 5 neighbors that are not leaves, the answer is 3.
Proof1:
Let's call this vertex v, let's call 5 neighbors u_1, u_2, u_3, u_4, u_5.
Let's prove that before visiting v we can visit at most 2 of u's.
It would also mean that after visiting v we can also visit at most 2 of u's.
This means that we cannot visit all 5 u's.
So, let's assume that we start in the subtree of u_1. Before leaving this subtree we had to finish at u_1 itself, otherwise we could jump only to v, we don't want that yet.
So, we jump from u_1 to u_2. We somehow traverse the subtree of u_2 and finish in some of it's children. Now we cannot go to the subtree of u_3 without passing by v.

This means that all interesting vertices have 3 or 4 neighbors that are not leaves.

Lemma2: If vertex v has 4 neighbors that are not leaves, the traversal must look like:

- We traverse the subtree of u_1 finishing at u_1.
- We traverse the subtree of u_2 starting at u_2 and finishing at some of it's children.
- We go to v.
- We traverse the subtree of u_3 starting at some of it's children and finishing at u_3.
- We traverse the subtree of u_4 starting at u_4 and finishing at some of it's children.

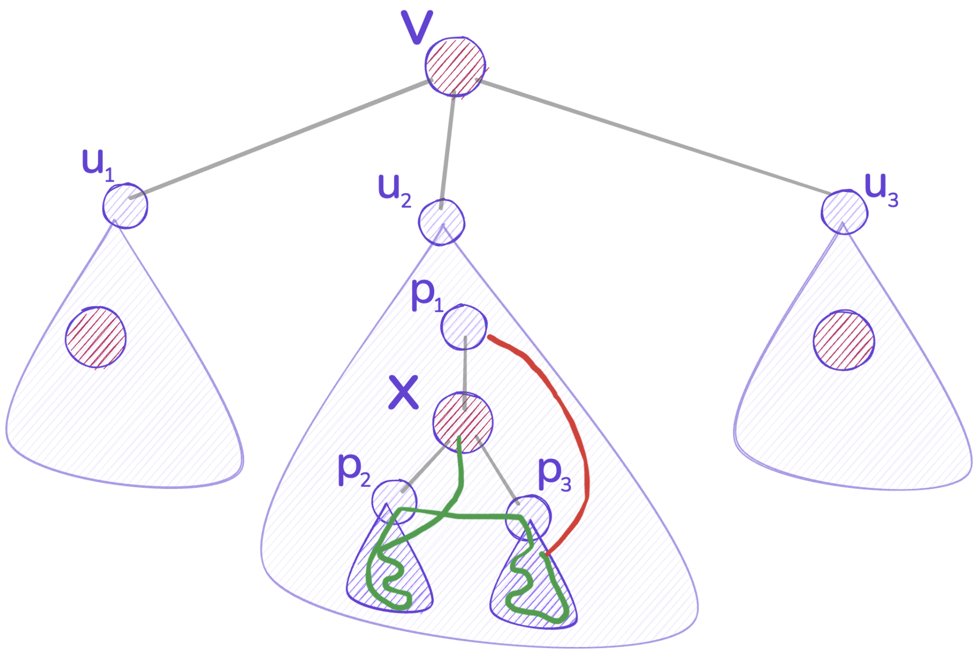
(Also there are leaves, but we can cover them between jumping from u_1 to u_2.)



Proof2:
Lemma1 proves it, we need to visit u_1 and u_2 before v, u_3 and u_4 after v, that's the only way to do that.

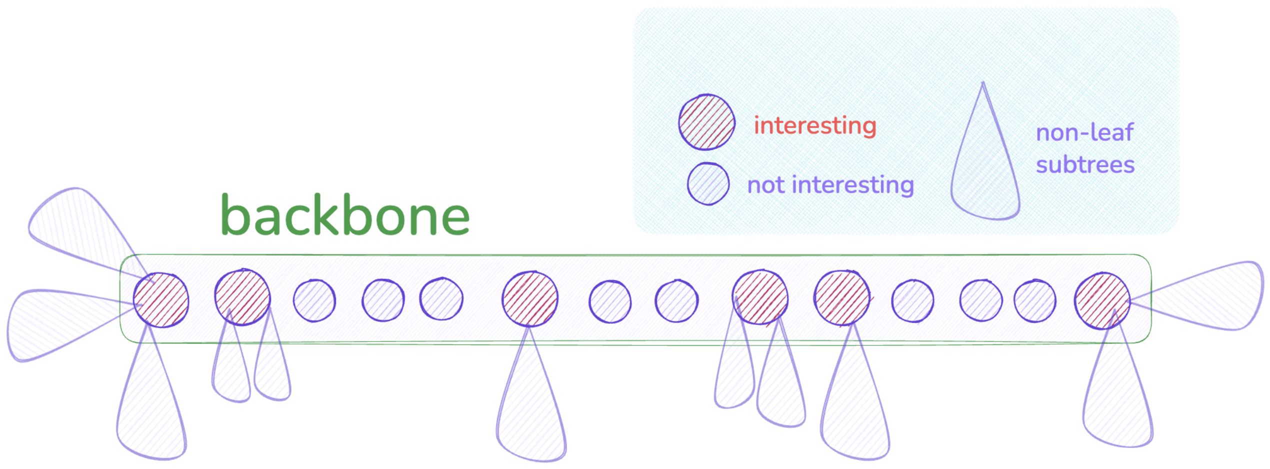
Lemma3: There should be a simple path(bamboo) that contains all interesting vertices, otherwise the answer is 3.
Proof3:
Let's prove by contradictions.
Let's assume that there is no simple path that contains all interesting vertices.

This means that there is some interesting vertex v that has interesting vertices in 3 of it's children subtrees.
Let's call these children u_1, u_2, u_3.
Let's say that we start in the subtree of u_1. Let's say that the second subtree we visit is the subtree of u_2.
Let's call the interesting vertex in the subtree of u_2: x.



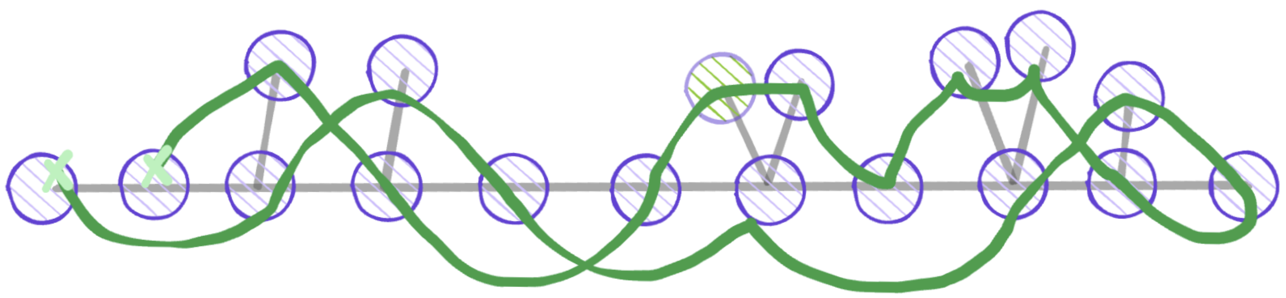
x has 3 neighbors that are not leaves, let's call them p_1, p_2, p_3, where p_1 is the parent of x.
Let's look at the moment when we enter the subtree of x. We either start at x or jump from p_1 to some p_i. WLOG, we start at x.
We start at x, we have to go to the subtree of p_2, we must finish at p_2, then we jump to p_3, we traverse the subtree of p_3, then we want to leave the subtree of p_3, but we cannot do that, because we need to jump from some children of p_3 to p_1, but the length of this jump is 3.

Let's call the smallest simple path that contains all interesting vertices a *backbone*.



The ends of the backbone are interesting vertices, otherwise we could make it shorter.
Let's call a tree *simple* if it doesn't have interesting vertices.
On the image, all non-leaf subtrees are simple by construction.
The only structure for a simple tree is a bamboo + some leaves.

Lemma4: We can traverse a simple tree starting at its end and finishing at this end's neighbor.
Proof4:



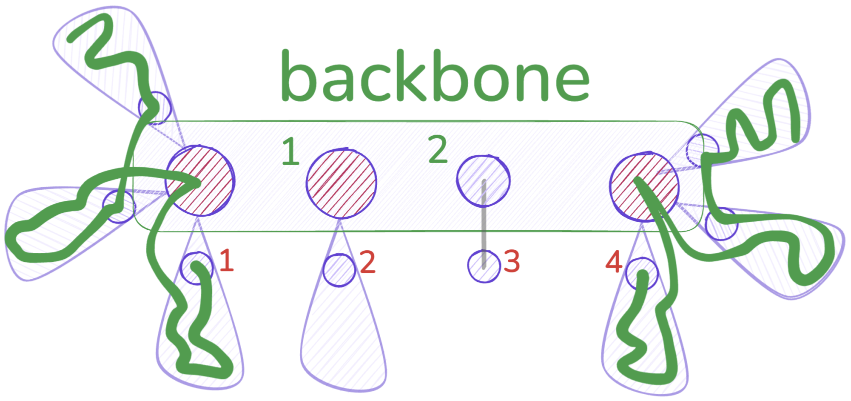
We jump by odd ones to the left, then by even ones to the right. We go over leaves of a vertex in the middle of jumping over it along the bamboo.

Sowe can compress every non-leaf subtree into just a bamboo of two vertices, since their traversals are equivalent in a sense.

If the length of the backbone is 1 or it doesn't exist at all (there is no interesting vertex), we already know how to traverse. So, let's work with a backbone of size at least 2.

Lemma5: We should start in one of the non-leaf subtrees of the leftmost vertex of the backbone (and finish in one of the non-leaf subtrees of the rightmost vertex of the backbone).
Proof5:
The proof is similar to Proof3.
If we start in the middle, by the moment we approach the end point of the backbone we should have traversed everything besides 2(or 3) non-leaf subtrees that it has, because we will not be able to come back. This means that we finish in one of the non-leaf subtrees of this end. But there are 2 ends, so we should start "at" one and finish "at" the other.

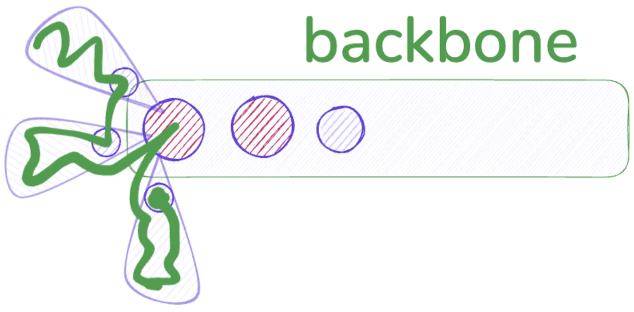
Lemma6: If we have two interesting vertices(u and v) that have 4 non-leaf neighbors and there's no vertex that has exactly 2 neighbors(**Note**: exactly 2, not just 2 non-leaf ones) on the backbone path between them, then the answer is 3.
Proof6:
So, we have some segment of the backbone that starts and ends with vertices with 4 non-leaf neighbors and between them all other backbone vertices have at least one extra neighbor beside the backbone neighbors.
Let's draw the traversal of non-leaf subtrees of the endpoints. We would finish the traversal in one of their children by Lemma2.

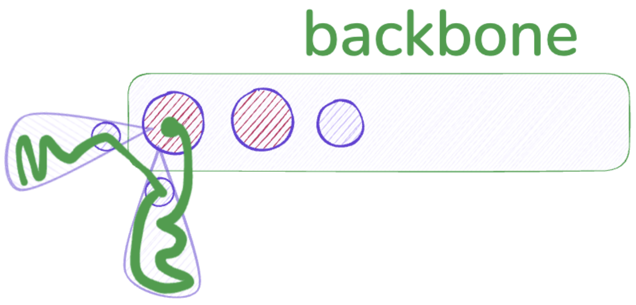


Now, we need to reach red number 4 from red number 1. This is impossible. We have 2 more red numbers than green ones and we need to have at least one green number between any two red ones (we cannot jump from a subtree of some backbone vertex to a subtree of another backbone vertex without visiting the backbone).

Lemma7: If the condition from Lemma6 doesn't hold, the answer is 2, we can just build the construction.
Proof7:
Let's traverse the graph in the following way: we are trying to traverse the prefix of the backbone (including all subtrees). If we can finish the prefix in the vertex of the backbone we do that, otherwise we are in some child of that vertex.
Let's prove that if we couldn't finish in the backbone vertex then it means that we had an interesting vertex with 4 non-leaf neighbors followed by some interesting vertices with 3 non-leaf neighbors.

Let's start with the induction base, in the picture below you can see two cases for the very first interesting point.
If it had 4 non-leaf neighbors, we finish in a child, otherwise in the backbone vertex itself.





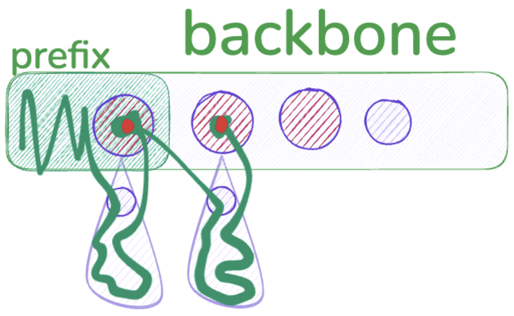
Then in the induction step we need to prove three things:

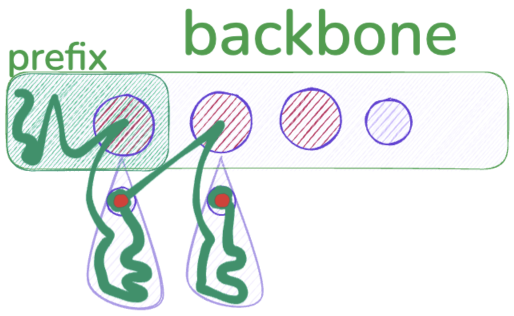
- If we finished the prefix in a child and the next backbone vertex has degree 2, then we finish the next prefix in that vertex.
- If we finished the prefix in the backbone vertex and the next backbone vertex is interesting and has 4 non-leaf neighbors, then we finish the next prefix in a child.
- All other cases don't change our state.

The proofs are:

- First is obvious, we just jump there and that's it.

- Second we know by Lemma 2. We couldn't finish the prefix in a vertex that has 4 non-leaf neighbors since that prefix contains 3 of them and we know that such a vertex is squeezed between 2 and 2.
- To prove the other cases let's look at the image below (you can see a subtree in the image but it works the same way if it's just a leaf).





So, a vertex with 4 non-leaf neighbors forces us to a child and a vertex of degree 2 saves us back to the backbone vertex.

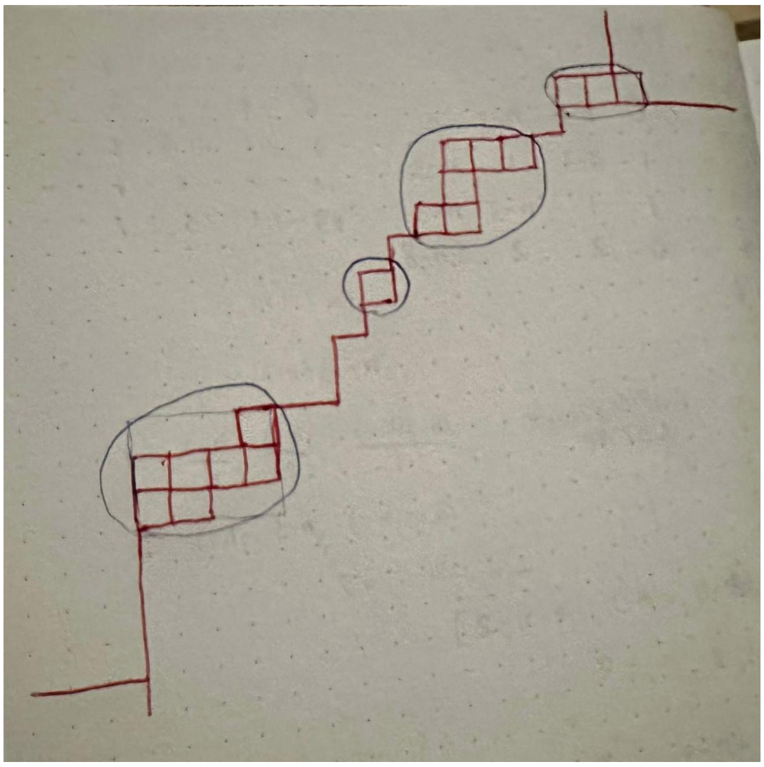
Final criteria in short

- There's no vertex with 5+ non-leaf neighbors.
- *Interesting*(3+ non-leaf neighbors) vertices form a path(called *backbone*).
- The backbone must contain a vertex of degree 2 between any two *interesting* vertices that have 4 non-leaf neighbors.

Problem M. Manhattan Graph

A rigorous proof is awaited.

Solution
Let's look at an example of such a graph and describe what we have to consider.



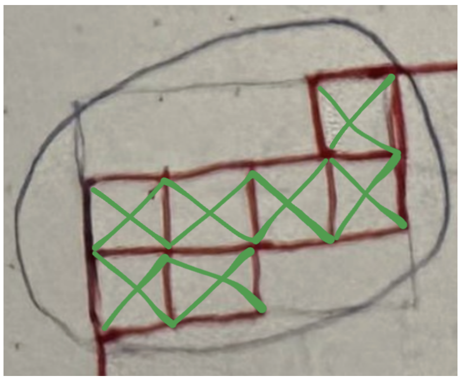
Any graph looks like this:

- We have several biconnected components(let's call them blocks), aligned up-and-right. They are circled on the image.
- Every biconnected component is a rectangular grid with some "corners" cut.
- Any two biconnected components are connected by a simple path. It must start in the top-right corner of the lower block and got to bottom-left corner of the upper block.
- We have some tails. Up to 4. We can have one tail from 4 side: bottom of the bottom-left block, left from the bottom-left block, same for top-right.
- bottom-left block instead of separate tails can have first a simple path starting at the bottom-left corner that diverges to two separate parts in a vertex of degree 3. (you can see such example on the image).

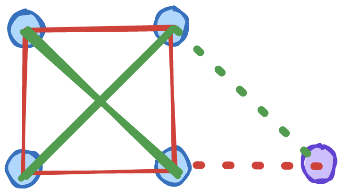
So, the task was just to handle it properly.

First, we need to detect the biconnected blocks (if two blocks are connected not by a path but just by a point, we consider them different blocks, though we can treat them as one).

We can create additional diagonal edges. Let's add a green edge between two vertices if they share two neighbors.



Now we can easily reconstruct the positions of all vertices of the block.
Let's start with one cycle of length 4. Let's take a side. If there is a vertex at the distance of 1 from the cycle, we can deterministically decide the coordinates of that vertex. Because it would be connected with one vertex by a red edge, with another one by a green one, and for two vertices there are only two possible positions if we know a green and a red edge and one of them (the mirror one) is already taken.



With that process, we can grow the component until it grows. This lets us reconstruct the whole block.

After that, we just need to proceed with handling the process described above. Check that all simple paths between blocks are correct, tails are correct, there's nothing extra.

Criteria
Once we built our solution, to check that it satisfies the required equality of distances we need to check that along any axis for any coordinate the vertices form a continuous segment. On the example below we can see one wrong coordinate (top x coordinate).