Problem A. Building Marble Tracks

Divide the line segments in the input into blocks of size $\sqrt{n}$. We will add these blocks one by one to the solution set S. The general plan to add a new block of segments T is:

1. Filter out segments in T that intersect with some segment already in S. This can be done using a modified version of the Bentley-Ottmann algorithm in $\mathcal{O}(n \log(n))$ time.
2. Perform pairwise intersection tests between the remaining segments in T. Because there are at most $\sqrt{n}$ elements in T, this can be done trivially in $\mathcal{O}(n)$ time.
3. Add all remaining line segments of T to the solution set S. This can be done trivially in $\mathcal{O}(\sqrt{n})$ time.

The filtering in the first step uses a linear sweep over the union of line segments already in the solution and the line segments in the new block. For this step, we modify the Bentley-Ottmann algorithm. The next paragraph describes this algorithm, you can skip it if you're already familiar with the Bentley-Ottmann algorithm.

The goal of the general Bentley-Ottmann algorithm is to find the intersection between any two line segments in a set of line segments. It performs a linear sweep over the 2D plane. The algorithm maintains all line segments that are currently intersecting the sweepline in the order that they do intersect it in (e.g. from bottom to top for a sweepline going left to right). To do so, there are three kinds of events: adding a line segment at its start, removing it at its end, and swapping the position of two line segments at their intersection. Because not all intersections can be calculated in advance, a line segment checks for the next intersection with the adjacent line segments in the sweep set when it is added or swapped. When a line segment is removed, the two line segments were next to it are now neighbors and are checked for their next intersection as well. These intersections are then added to a priority queue which maintains all events.

This algorithm $\mathcal{O}((n + s) \log(n))$ time, where s is the number of intersections between line segments. To have the first step run in $\mathcal{O}(n \log(n))$ time, it is left to show that $s \in \mathcal{O}(n)$:

- Intersections between line segments in the solution set:
  By the requirement of the task, these do not exist.
- Intersections between line segments in the new block:
  These line segments may intersect pairwise. However, there are only $\sqrt{n}$ line segments in the block and therefore only $\mathcal{O}(n)$ intersections.
- Intersections between line segments of different sets:
  These also may intersect pairwise for a total of $\mathcal{O}(n^{3/2})$ intersections. This can be reduced to $\mathcal{O}(\sqrt{n})$ by simply removing the line segment from the new block once the first intersection of this kind is found. This results in each line segment of the new block having at most one of this kind of intersection. Remember, we only want to know for each line segment in the block if it intersects some line segment in the solution set, not necessarily all intersections.

The dominating time factor for one block is therefore filtering the line segments of the new block with line segments already in set. Because there are $\sqrt{n}$ blocks, the algorithm runs in $\mathcal{O}(n^{3/2} \log(n))$ time.

Problem B. Build Well

First, we want to solve a simplified version of the problem: given some bricks is it possible to build one row. This problem is also known as the coin-change problem and can be solved in $\mathcal{O}(\frac{w \cdot n}{64})$ with bitsets or in $\mathcal{O}(w \cdot \log(w)^2)$ with fft ($\mathcal{O}(w \cdot \log(w))$ is also achievable). Surprisingly, both of these approaches are fast enough.

Now we can notice that if a solution that avoids bricks of length 1 exists, we are done. If such a solution does not exist and no bricks of length 1 are available, we are also done.

At this point, we can assume that blocks of length 1 are available. Now we find a solution for both rows simultaneously and we restrict ourselves to a solution where the second row has an offset of one to the first row. To do this we create gadget blocks that span two rows and have the following properties:

- the gadget has the same length in both rows
- the second row is offset by 1
- the gadget creates no gaps in itself
- the first row does not start with a 1 block and the second row does not end with a 1 block i.e. any pair of gadgets can be concatenated without creating gaps.
- a solution to the original problem exists if and only if a single row solution with the gadgets blocks exists.

We can now observe that it is sufficient to create minimal gadget blocks i.e. those that use only one non 1 block. For each block w_i of the original problem, we create $w_i - 2$ gadget blocks where the first row is a w_i block followed by some 1 blocks and the second row is some 1 blocks followed by a single w_i block. With these gadget blocks we can again solve the coin-change problem and check if it is possible to combine gadgets to a total length w, if yes this is also a solution to the original problem, if no there is no solution.

The only problem remaining is: how can we create all those gadget blocks? For this, we observe that a block of width $w > 1$ results in the gadget blocks of length $w, ..., w-2$. Therefore, we can use a difference array/prefix sums to find all lengths for which we can build a gadget block in $\mathcal{O}(w)$.

There also exist other – more greedy – solutions. of the original input

Problem C. Centrifuge

Lets first calculate the answer for one leaf l.

First, we root the tree at vertex l. Then we do a tree dp, where $dp[v]$ represents the expected liquid moving upwards from below vertex v towards its parent.

$$dp[v] = \frac{\sum_{u \in U} dp[u]}{max(1, deg[v] - 1)} + \frac{a[v]}{max(1, deg[v] - 1)} \cdot \frac{size[v] - 1}{n} + \frac{a[v]}{deg[v]} \cdot \frac{1}{n}$$

Where U are the children of vertex v, $deg[v]$ is the degree of vertex v, and $size[u]$ is the subtree size of u.

The sum has 3 parts: the liquid from below u, the liquid from u if the chosen center is below u, and the liquid from u if the chosen center is u. For the root, the last term needs to be subtracted again since, if it is chosen as center, the liquid will all flow downwards (except for $n = 1$).

To calculate the solution for all leaves, tree rerooting is needed.

Watch out as the initial water in each vertex could be larger than the modulo (especially for $n = 1$).

Problem D. Infinity Triples

Lemma: (n, a, b) is an infinity triple iff b is coprime to $\frac{n}{\gcd(a,n)}$

Proof: The triple (n, a, b) is an infinity triple iff the equation

$$\sum_{j=0}^{k-1} a \cdot b^j = a \cdot \frac{b^k - 1}{b - 1} \equiv 0 \pmod{n}$$

has infinitely many positive integer solutions k. Now, we want to get rid of the factor a on the left side. Let $g = \gcd(n, a)$. Then, the above is equivalent to

$$\frac{a}{g} \cdot \frac{b^k - 1}{b - 1} \equiv 0 \pmod{\tfrac{n}{g}}$$

Since $\frac{a}{g}$ and $\frac{n}{g}$ are coprime, we can multiply both sides by $(\frac{a}{g})^{-1}$. Multiplying both sides by $b-1$, we get the equivalent congruence

$$b^k - 1 \equiv 0 \pmod{\tilde{n}}$$

where $\tilde{n} = (b-1)\frac{n}{g}$. If b and $\tilde{n}$ are not coprime, the equation has no solution. Otherwise, there are infinitely many solutions (one way to see this is Euler's theorem, every multiple of $\varphi(\tilde{n})$ is a solution). Thus, (n, a, b) is an infinity triple iff

$$1 = \gcd\left(b, (b-1)\frac{n}{g}\right) = \gcd\left(b, \frac{n}{\gcd(a, n)}\right)$$

We know three solutions for enumerating the triples with the above characterization:

Solution 1:

- Fix $b = p_1^{e_1} \cdot \ldots \cdot p_\ell^{e_\ell}$ and iterate it over all values $\le m$.
- Iterate over all values $s = p_1^{f_1} \cdot \ldots \cdot p_\ell^{f_\ell} \le b$ which only contain prime factors from $p_1, \ldots, p_\ell$. We interpret s as the part of n which contains only prime factors of b.
- Count the number of a, n such that
  1. $s|a$ and $a < b$
  2. $s|n$ and $\frac{n}{s}$ is coprime to b

  The number of a is simply $\lfloor \frac{b-1}{s} \rfloor$ and the number of n can be computed using PIE in $\mathcal{O}(2^\ell)$.

Solution 2:

- Let x_d denote the number of triples (n, a, b) with $1 \le n \le m$, $1 \le a < b \le m$ and $\gcd(b, \frac{n}{\gcd(a,n)}) = d$.
- Then, $x_d = y_d - \sum_{k=2}^{\lfloor m/d \rfloor} x_{dk}$ where y_d is the number of triples with $d \mid \gcd(b, \frac{n}{\gcd(a,n)})$.
- A triple is only counted by y_d when $d \mid n$. So let's iterate over all pairs $d \mid n$.
- A value a is valid iff $d \mid \frac{n}{\gcd(a,n)}$. If p is a prime divisor of d and p^e is its highest power dividing n, then the condition on a implies $p^{e+1} \nmid a$.
- As in solution 1, we can use PIE to count the number of valid a, b. To do this, we need to be able to compute the number of a, b such that $t \mid a,\ d \mid b,\ a < b$ for some t efficiently. To do this, note that there are $\lfloor \frac{m}{d} \rfloor \lfloor \frac{m}{t} \rfloor$ pairs without the condition $a < b$. Let's subtract the bad pairs with $a \ge b$ to obtain

  $$\left\lfloor \frac{m}{d} \right\rfloor \left\lfloor \frac{m}{t} \right\rfloor - \sum_{k=1}^{\lfloor m/t \rfloor} \left\lfloor \frac{kt}{d} \right\rfloor$$

  It is a well known problem to compute this in logarithmic time. Two possible ways to do it are:

  – Find the convex hull of the points under the line using continued fractions and apply Pick's theorem.
  – A simpler solution: https://codeforces.com/blog/entry/65500?#comment-496162

Solution 3:

- Fix n and iterate it over all values $\leq m$.
- Iterate over all subsets S of prime factor of n. Those will be the primes which n and b have in common.
- Use PIE to calculate the number of (a, b) with $a < b$ such that b shares exactly the prime factors of S with n. To calculate this, it is again necessary to calculate a sum of floors efficiently, similarly to solution 2.

For the solutions (in particular the first), it is difficult to estimate the asymptotic time complexity because it involves sums of functions depending on the distribution of primes. The easiest way to verify that the solutions are indeed fast enough is to implement them or to estimate the number of operations using a simple program.

Problem E. Taxi

Let T be the given tree. We want to find the shortest path on the graph G' where edge (u, v) has weight $a_u + b_u \cdot d(u, v)$ where $d(u, v)$ is the distance of u and v in T. This graph has too many edges to do this naively.

To optimize this, we use the following properties:

Let's look at another simpler version of this problem. Imagine there are different options for types of taxis at vertex 1 and no taxis at any other node. Each taxi's cost function can be represented as a linear function. Given the distance to another vertex (i.e. with LCA), this cost can be calculated efficiently using the convex hull trick.

Going back to the original problem, we can take advantage of some useful properties of the centroid decomposition trees. Let T' denote the centroid decomposition tree of T. For any two vertices u and v in T, either v is a descendant of u in T', or at least one of u's ancestors lies on the simple path between u and v. Furthermore, the sum over all descendants and ancestors is bound by $\mathcal{O}(n \cdot \log n)$ as T' only has depth $\mathcal{O}(\log n)$.

Using the centroid decomposition tree, we could maintain the minimum cost from any vertex to all ancestors without changing taxis. We save these costs in a convex hull datastructure at the ancestors. We can now find the minimum cost from any already processed vertex to any new vertex without changing taxi by just checking all of its descendants and ancestors.

We can also further simplify the problem by noticing that we will only change to taxis with lower cost per distance traveled. In other words, for the minimum cost, the cost per distance traveled will be strictly decreasing.

Using all of the properties above, we can start solving the actual problem. First, we iterate over the vertices u in non increasing order of cost per distance. For this node, we add its cost function to all ancestors (of T') including itself. The cost function will be the sum of the minimum cost to get to u and the base cost a_v with slope b_v. To find this minimum cost, one only needs to check the cost from already processed descendants of v and through all ancestors.

Then we can iterate over all vertices and calculate the minimum distance by repeating this process. Adding the cost function will not be necessary anymore as any taxi you might take will already have been added.

This process is correct since the first iteration calculates the minimum cost to all vertices u that do not use any more expensive cost per distance traveled than b_u. While this is not the actual minimum cost to get to vertex u, this is the minimum if it would be optimal to change taxis at this vertex.

In the second iteration, the final answer will be calculated since all vertices where taxis will be changed are already processed.

In total, the time complexity is $\mathcal{O}(n \log^2 n)$ because $\sum_u$(Number of ancestors and descendants) $= \mathcal{O}(n \log n)$ and the other data structure operations take $\mathcal{O}(\log n)$ time.

Problem F. Periodic Sequence

We reduce a and b to their minimal periodic form. For example: $abcabcabc \to abc$ and $bcabca \to bca$. This can be done with kmp or z-function. For kmp, the cycle length is $\begin{cases} n - kmp[n-1] & \text{, if that value divides n} \\ n & \text{, else.} \end{cases}$

After this, we just need to check if a and b are (cyclically) equal. This can again be done with kmp or z-function by checking if $a'a'$ contains b'.

Problem G. Scheduling

First of all, we want to perform coordinate compression because there is no constraint on the sum of the maximum r_i over all testcases. To do this, we construct a set S: iterate over all left endpoints ℓ_i and add the three smallest $x \geq \ell_i$ to S which are not already contained in S. Then, we replace $[\ell_i, r_i)$ with $[\ell_i', r_i')$ such that $\ell_i' = \#\{x \in S : x < \ell_i\}$ and $r_i' = \#\{x \in S : x < r_i\}$. We claim that it is equivalent to solve the problem on those compressed intervals. To prove this, one can consider a schedule with lexicographically minimal meeting times and verify that the meeting times are a subset of S. So from now on, we can assume that $\ell_i, r_i \in \mathcal{O}(n)$.

Instead of directly constructing the schedule, we construct a set B of possible meeting times which contains no consecutive points of time. We want to find B such that the intervals $[\ell_i, r_i)$ can be matched to points from B which are also in the interval. By using Hall's theorem, one can observe that B is valid iff the following condition (*) is satisfied:

$$\text{For every integer interval } I = [\ell, r) \text{ we have: } \#(B \cap I) \geq \#\{[\ell_i, r_i) \subseteq I\}$$

So for every interval $I = [\ell, r)$ there is a number $f(I) = \#\{[\ell_i, r_i) \subseteq I\}$ meaning that B must contain at least $f(I)$ elements of I. We also write $d(\ell, r) = d(I) = \#(B \cap I) - f(I)$ and thus we want to achieve $d(I) \geq 0$ for all I. If we can find such B, we can find the matching greedily in $\mathcal{O}(n \log n)$. Otherwise, there is no answer. So let's try to find a valid B.

Define the balance of interval $I = [\ell, r)$ as $\phi(I) = \phi(\ell, r) = (r - \ell) - 2f(I)$.

- If there is an interval with $\phi(I) < -1$, then there is no answer.
- If all intervals have balance ≥ 0, we can use $B = \{0, 2, 4, \ldots\}$.
- If $\phi(\ell, r) = -1$. Every valid B must contain $\{\ell, \ell+2, \ell+4, \ldots, r-1\}$. Let F denote the set of values which B is forced to contain because of this condition.

For efficient implementation of the above observations, we can use a minimum segment tree with range addition updates and iterate r. Then index ℓ of the segtree maintains $\phi(\ell, r)$. For every r we do a minimum query to check the above cases and if there is $\phi(\ell, r) = -1$, we find the smallest such ℓ. With this we can recover F in $\mathcal{O}(n \log n)$.

From now on, assume the last case because we are done in the first two cases. We need to add some integers to F to construct a valid B. We claim that the following general strategy is optimal:

(i) For the smallest $x \in F$, add $x-2, x-4, x-6 \ldots$ and for the biggest $x \in F$ add $x+2, x+4, x+6, \ldots$

(ii) For two consecutive (in sorted order) elements $p, q \in F$, try to add as many elements as possible which come from $[p, q]$. If $q-p$ is even, this means that we add $p+2, p+4, \ldots, q-4, q-2$. Otherwise there is a gap $g = g(p, q)$ and we add $p+2, p+4, \ldots, g-2, g+1, g+3, \ldots, q-4, q-2$.

To prove this, one considers a solution which does not satisfy those conditions. If (ii) is not satisfied, we can find a structure like `001010100`, where we represent B as bitset. We can change this to `010101010`. It

is impossible that some interval satisfied (*) before and violates it now, because it would have $\phi(I) \leq -1$. For (i) we can use a similar argument.

Now, we assign values for the gaps from left to right. For $g(p, q)$ we greedily choose the minimum value such that $d(I) \geq 0$ is satisfied for all $I = [\ell, r)$ with $r \leq q$. This can be implemented efficiently using a sweep line with a segment tree. We iterate r and index ℓ of the segment tree maintains $d(\ell, r)$. By using a minimum query, we can check if it is allowed to have a gap at the current r.

The total time complexity is $\mathcal{O}(n \log n)$

Problem H. Mod Graph

For every query, we want to find a path through G that sets the counter of every vertex to zero. To do so, we will construct a new graph G' which has the same set of vertices and directed edges. This new graph will have an Euler Cycle. Walking along the edge (u, v) in G' then corresponds to walking along the edge $\{u, v\}$ in G.

For every undirected edge $\{u, v\}$ in G, we add $b_u \cdot b_v$ of both the edges (u, v) and (u, v) to G'. The graph G' is now connected and every vertex has the same in and out degree, so an Euler Cycle exists. Because every vertex v is entered $deg(v) \cdot b_v$ times, this does not modify the values in the vertices. We can now increment the value a_u and a_v of two adjacent vertices u and v by 1 by adding one copy of (u, v) and (v, u) to G'.

Next, observe that for any vertex u, we can add b_u to any vertex other than u. With the previously shown gadget, this is clear for all neighbors v of u: Just add 1 to both u and v a total of b_u times, and because u's counter calculates modulo b_u, its value remains unmodified. Now assume that we already know how to add b_u to the counter of some vertex v, but not one of its neighbors w. We can add b_u to a_w by first adding b_u to both a_v and a_w by just adding edges between v and w, and then repeatedly adding b_u to v until it is back at its original value.

With the ability to add b_u to any other vertex v, we can see that all b_u can be replaced by $b := \gcd_{v \in V(G)} b_v$ because of Bezout's lemma.

We can make almost all vertices zero (except for one): Pick a vertex t, walk from s to t and build a DFS-tree from it. When we finish vertex $u \neq s$ with parent p, we add a value to the edge $\{p, u\}$ such that $a_u = 0$. After that, all vertices except t are guaranteed to be zero.

If G contains an odd cycle $u_1, \ldots, u_k$, we can walk around the cycle once and then subtract 1 from every edge of the form $\{u_{2i}, u_{2i+1}\}$. This means we added 1 to a_{u_1} and no other value changed. If we choose u_1 as the root of our DFS-tree of the previous paragraph, this allows us to make all values zero. Thus, the answer is always yes if the graph is not bipartite.

If the graph is bipartite, the following invariant is maintained during our walk modulo b: $I = (R - B) + c$ where R is the sum of all a_u for red vertices u, B is the sum of all a_u for blue vertices u, and c is 0 if we are currently on a red vertex in our walk and 1 otherwise. This way, when walking to a red vertex, we increment R and decrement c, leaving I unmodified. When walking to a blue vertex, we increment B (which decrements I) and decrement c. This means that the value of I is only dependent on the starting values of all counters. We can easily maintain I through every update query in $\mathcal{O}(1)$ time.

Remember that starting at vertex s increments a_s by 1, which modifies I for this query only. You may also choose not to make the last step in the Euler Cycle of G', which would undo this increment.

Last, there is one exception to the bipartite case: If the graph has only one vertex, we cannot walk at all. In this case, we just output `YES` if $a_v + 1 = b_v$, and `NO` otherwise.

Problem I. In the Treetops

Before we get to more complex cases, note that an outerplanar graph with only one biconnected component has a trivial Hamiltonpath by traversing along the outer face.

There are two conditions related to cut vertices this graph has to fulfill:

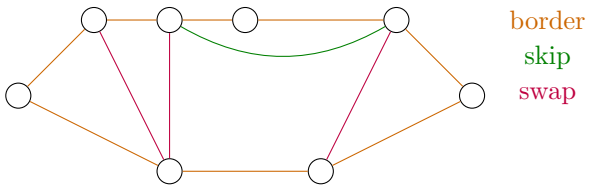
- If the removal of any cut vertex results in three or more seperate components, there is no Hamiltonpath. This is because every possible simple path starts in one of these components (or the cut vertex) and ends in another (or the cut vertex) and can therefore never visit all three components.
- If any biconnected component has three or more cut vertices, there is no Hamiltonpath. This has a similar justification to the condition above.

Therefore, the cut-vertex-block-tree has to be a path. This check requires the biconnected components, which can be calculated in $\mathcal{O}(n)$ time. The biconnected components are also required for later steps.

The biconnected components forming the endpoints of this path may have any form, the Hamiltonpath can be found the same way as in the trivial case where one endpoint is on the cut vertex. For all other biconnected components, we now have to find a Hamiltonpath with a dedicated start- and endpoint. To do so, first divide all edges into three sets:

- border edges: edges on the outer face.
- skip edges: non-border edges with both ends on the same (lower/upper) border
- swap edges: edges that are not border edges and not skip edges

Below is an example, where the left- and rightmost vertex are the cut vertices that are to be connected by the Hamiltonpath:



Although the Hamiltonpath is not directed, it is helpful to think about building the path from the left to the right cut vertex. Note that traversing a border edge in the opposite direction (i.e. right to left) makes it impossible to reach the right cut vertex. Further, taking a skip edge forces one to traverse a border edge in such a way, therefore we can not take any of them either.

On the remaining possible edges, find a path from the left to the right cut vertex that alternates between taking a border and a swap edge. This can be done in $\mathcal{O}(n)$ if such a path exists. The Hamiltonpath can be constructed from the alternating path by using the same swap edges and the complement of the border edges. Because these two paths are complementary in some sense, the Hamiltonpath exists exactly if the alternating path exists.

Alternatively, there is a solution using dynamic programming: We now consider partial Hamiltonpaths starting at the left cut vertex that cover some prefix of the upper vertices and some prefix of the lower vertices. Our states are now the last covered vertex on the top border, on the bottom border, and whether our current endpoint is on the top or bottom border. The number of states is now quadratic, but it can be reduced to linear by realizing that all states with a possible choice are ones where the vertex on the upper and lower border are connected by an edge.

Problem J. Permutation Recovery

Let a_{ij} denote the entries of the matrix. For every $a_{ix} = y$, there is a corresponding entry $a_{jy} = x$ (because the original matrix contains the inverse of every permutation).

We create a multi-graph G with an undirected edge $\{x, y\}$ for each corresponding pair. Note that G may contain self-loops.

Every vertex has the even degree $2k$, so we can find an Euler tour on each component of the graph. This Euler tour assigns a direction to every edge such that every vertex has k outgoing and k incoming edge endpoints.

For every vertex u, split it into u_{in} and u_{out} in an auxiliary bipartite graph G'. If the Euler tour directed an edge from u to v, add the edge $\{u_{out}, v_{in}\}$ to G'. Graph G' is k regular, so it has a perfect matching by a corollary of Hall's theorem. We can find this perfect matching in $O(nk\sqrt{n})$ with Hopcroft-Karp.

We create a permutation π where if u_{out} is matched with v_{in}, we set $\pi(u) = v$. For every i, remove one edge $\{i, \pi(i)\}$ from G, which leaves a $(2k-2)$ regular graph and repeat the same algorithm for the remaining $k-1$ permutations.

This leads to a $O(k^2 n\sqrt{n})$ solution.