

Problem A. Letters from Parts

Since horizontal sticks are only used in the letter A, the number of horizontal sticks h must equal the number of A's. If the number of diagonal sticks is less than $2h$, then it is impossible to assemble all elements into letters. Otherwise, we have $d - 2h$ diagonal sticks remaining. All of them must be used in letters N, so if the number of vertical sticks is less than $2(d - 2h)$, then it is also impossible to assemble. Otherwise, we have h letters A, $d - 2h$ letters N, and $b - 2(v - 2h)$ letters I. The answer will be the minimum of these three numbers (since each "NAI" string requires one letter of each type).

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 $w, \dots, w - 2$. 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.

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

Where U are the children of vertex v , $\deg[v]$ is the degree of vertex v , and $\text{size}[u]$ 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

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

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 $b = p_1^{e_1} \cdot \dots \cdot p_\ell^{e_\ell}$ and iterate it over all values $\leq m$.
- Iterate over all values $s = p_1^{f_1} \cdot \dots \cdot p_\ell^{f_\ell} \leq b$ which only contain prime factors from $p_1, \dots, p_\ell$. 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 $1 \leq n \leq m$, $1 \leq a < b \leq m$ 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 $d | \gcd(b, \frac{n}{\gcd(a, n)})$.
- A triple is only counted by y_d when $d | n$. So let's iterate over all pairs $d | n$.
- A value a is valid iff $d | \frac{n}{\gcd(a, n)}$. 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 $a \geq b$ 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 $\sum_u (\text{Number of ancestors and descendants}) = \mathcal{O}(n \log n)$ 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: $abcabcabc \rightarrow abc$ and $bcabca \rightarrow bca$. This can be done with kmp or z-function. For kmp, the cycle length is
$$\begin{cases} n - \text{kmp}[n - 1] & , \text{ if that value divides } n \\ n & , \text{ else. } \end{cases}$$

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. Operations on Subarrays

How will the solution work for the entire sequence?

We count the bits at each position. Since the operation allows us to swap bits (0 or 1) between any two numbers in the sequence, we can construct a solution starting from a bit frequency vector, optimally distributing the available bits.

Let w be the maximum number of bits for any value in the sequence (in the problem statement, $w \leq 20$).

We start with all values set to 0. We consider the bits from the most significant to the least significant, following this principle: at each position, we assign 1 bits to the smallest values constructed so far, and 0 bits to the largest values, then we re-sort the sequence (complexity $\mathcal{O}(w \cdot n \log n)$) or merge it (complexity $\mathcal{O}(w \cdot n)$).

It can be proven that this strategy leads to an optimal solution using a simple argument that we will elaborate on later.

To achieve better runtime, we notice that after processing each of the w bits, the number of distinct values in the sequence increases by at most 1. Therefore, we can "simulate" the naive algorithm described above on a "compressed" representation of the sequence defined by pairs (value, frequency). Thus, the complexity of the algorithm becomes $\mathcal{O}(w^2 \log w)$ or $\mathcal{O}(w^2)$, depending on the implementation.

In the end, we will have at most w pairs (value, frequency), and the sum can be easily computed.

To answer a query, we compute the difference between the counters at the ends of the requested interval (as partial sums) and obtain the counter for the subarray for each of the w bits. We then apply the described algorithm.

It remains to justify the correctness of this algorithm.

It is clear that if we apply the operation between two numbers at a position where the bits are identical, the operation does not change anything. Thus, the only significant changes occur when swapping a 0 bit with a 1 bit.

Details of the Operation

The operation "moves" a value (power of 2) from one number to another. Initially, this transition is beneficial when the resulting values are closer to each other. This can be proven with a very simple mathematical calculation.

Reapplying the operation between the same two numbers allows us to swap entire sequences of bits between them.

Using this idea, we make an observation that fully justifies the solution:

Consider the optimal solution. We select two elements from the sequence. We divide their binary representations into suffixes and prefixes of equal length (the sum of which is clearly equal to w). It can be proven (again with a simple mathematical calculation) that it is optimal for the larger prefix to correspond to the smaller suffix.

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 (v, u) 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 $u_1, \dots, u_k$, 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. Arithmetic Progression and Triangle

The sum of the angles is equal to 180 degrees, so we have $a + (a + d) + (a + 2d) = 180$, so $(a + d) \cdot 3 = 180$ and $a + d = 60$, so the middle angle is equal to 60. If the given angle is greater than 60, the answer is -1 (it can't be minimal), otherwise the angles are uniquely determined and the answer is 60, 60 and $120 - a$, $120 - a$.

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.