

## Problem A. Assembling Strings from a Trie

The number of reconstructed strings must equal the number of leaves in the given trie. For any node  $x$ , all strings in its subtree occupy a contiguous interval in lexicographical order, since they share the prefix represented by  $x$ . This allows us to assign edge labels in a bottom-up manner.

When processing a node  $x$  with children  $y_1, \dots, y_k$ , assume that each subtree has already been optimally reconstructed. Let  $\text{seq}(y)$  denote the lexicographically sorted list of suffixes produced by the subtree rooted at  $y$  (excluding the common prefix up to  $y$ ). We assign labels  $1, 2, \dots, k$  to the edges from  $x$  to its children and choose an ordering of the children such that

$$\text{seq}(x) = 1 \cdot \text{seq}(y_1) + 2 \cdot \text{seq}(y_2) + \dots + k \cdot \text{seq}(y_k)$$

is lexicographically minimal. Here,  $c \cdot \text{seq}$  denotes prefixing every string in the sequence by character  $c$ , and  $+$  denotes concatenation.

The optimal ordering can be determined by pairwise comparisons. For two children  $u$  and  $v$ , placing  $u$  before  $v$  is better if and only if

$$1 \cdot \text{seq}(u) + 2 \cdot \text{seq}(v) < 1 \cdot \text{seq}(v) + 2 \cdot \text{seq}(u).$$

- If  $\text{seq}(v)$  is a **prefix** of  $\text{seq}(u)$ , then  $u$  should come first, and vice versa.
- Otherwise, the subtree with the lexicographically smaller sequence comes first.

This comparison reduces to a recursive comparison of the two subtrees: we compare their children in order, and if one sequence ends earlier, it is treated as the prefix case. Each comparison takes  $O(\min(\text{size}_u, \text{size}_v))$  time. By applying a small-to-large merging strategy, the total time complexity is  $O(m \log^2 m)$ , where  $m$  is the number of trie nodes.

Bonus. Can this be optimized to  $O(m \log m)$  time?

## Problem B. Budgeted Broadcasts

Fix a pair of vertices  $(a, b)$  at distance  $d$ . Its contribution to the total sum is

$$d \sum_{S \supseteq \{a, b\}} |S| = d \sum_{k=0}^{n-d-1} \binom{n-d-1}{k} (k+2) = d \cdot 2^{n-d-2} (n-d+3).$$

Here, we exclude the  $d+1$  vertices on the path from  $a$  to  $b$  from the set  $S$ .

Thus,

$$\text{Ans} = \sum_{a < b} d \cdot 2^{n-d-2} (n-d+3) = 2^{n-2} ((n+3)A - B),$$

where

$$A = \sum_{a < b} d \cdot 2^{-d}, \quad B = \sum_{a < b} d^2 \cdot 2^{-d},$$

and  $d$  denotes the distance between  $a$  and  $b$ .

It suffices to compute these sums by fixing one endpoint  $a$  as the root and aggregating contributions over all other vertices.

Let  $f_{u,k}$  denote the sum of  $d^k 2^{-d}$  over all vertices in the subtree of  $u$ , where  $d$  is the distance from  $u$ . Consider a child  $u$  of  $v$ . The contribution  $t_k$  of  $f_{u,k}$  to  $f_{v,k}$  is given by

$$\begin{aligned} t_0 &= \frac{1}{2} + \sum 2^{-(d+1)} = \frac{1}{2} (f_{u,0} + 1), \\ t_1 &= \frac{1}{2} + \sum (d+1) 2^{-(d+1)} = \frac{1}{2} (f_{u,0} + f_{u,1} + 1), \\ t_2 &= \frac{1}{2} + \sum (d^2 + 2d + 1) 2^{-(d+1)} = \frac{1}{2} (f_{u,0} + 2f_{u,1} + f_{u,2} + 1). \end{aligned}$$

By computing all values  $f_{u,k}$ , and applying standard add-subtract adjustments during re-rooting, we can obtain the required sums for every root.

Therefore, the problem can be solved in linear time. Alternatively, one may also directly apply tree DP to compute the sum over all pairs within each subtree.

## Problem C. Call for Papers

The final expected impact factor of a paper depends only on its first submission. For each conference, we first determine where a paper will be transferred if it is rejected, according to the given resubmission rules. Based on this, we compute, for each conference, the expected impact factor of the conference that will eventually accept the paper, assuming it is the first submission. This takes  $O(n^2)$  time.

We then perform a sweep line, process all submissions and deadline events in chronological order, and maintain a list of conferences currently accepting submissions. This set is ordered by impact factor. At any moment, the conference with the maximum impact factor in the set is the optimal choice for the first submission.

The overall time complexity is  $O(n^2 + m \log n)$ .

## Problem D. Dune

For each sandworm  $i$ , we compute the time interval  $[t_{l,i}, t_{r,i}]$  during which it passes through  $x$ .

We perform a sweep line over time. During the sweep, we maintain a multiset containing the lengths of all sandworms currently passing through  $x$ . It suffices to check only around event times. The time complexity is  $O(n \log n)$ .

## Problem E. Expected Inversions

The expected number of inversions can be written as the sum, over all ordered pairs of distinct elements  $(x, y)$ , of the probability that  $(x, y)$  forms an inversion.

Consider a fixed pair  $(x, y)$  with  $x \neq y$ , we only care which of  $x$  and  $y$  is placed earlier in the sequence. Suppose the sum of remaining elements at that moment is  $sum$ . Then  $x$  is selected with probability  $\frac{x}{sum}$  and  $y$  with probability  $\frac{y}{sum}$ . Conditioning on the first appearance of either  $x$  or  $y$ , the probability that they form an inversion equals  $\frac{\max(x,y)/sum}{x/sum+y/sum} = \frac{\max(x,y)}{x+y}$ , which is independent of the current state of the sequence. Therefore, the problem reduces to computing  $\sum_{i \neq j} \frac{\max(b_i, b_j)}{b_i + b_j}$ .

We solve this using D & C on the *values*. For  $l \leq i < m \leq j < r$ , we sum up

$$\sum_{k=l+m}^{m+r-2} \frac{1}{k} [x^k] \left( \sum_{i=l}^{m-1} cnt_i x^i \right) \left( \sum_{i=m}^{r-1} icnt_i x^i \right)$$

This can be done with FFT, and the total time complexity is  $O(V \log^2 V)$ , where  $V = \max b_i$ .

## Problem F. Finding Cyclic Isomorphism

For each pattern string  $S_i$ , we insert  $S_i + S_i$  into a generalized suffix automaton. For each cyclic representation of  $S_i$ , we locate its state in the automaton by binary lifting on the fail tree and record the length of  $S_i$  at that state.

For a query interval  $[l, r]$ , we determine the valid length interval of the prefix and suffix parts by computing the LCP of  $T[l, \cdot)$  and  $T[r, \cdot)$ , using binary search with hashing or suffix array. After fixing this interval, the problem reduces to counting valid middle substrings whose lengths fall within the allowed range.

To locate the automaton state corresponding to  $T[l, y - 1]$ , we process all queries offline, sorted by  $y$ , and match the text string  $T$  against the generalized suffix automaton in a single linear scan. Inserting  $T$  into the automaton is avoided to reduce memory usage.

Finally, we compute the total contribution of valid pattern lengths along the path from a node to a given ancestor in the fail tree. The total time complexity is  $O(N \log N + Q + M \log N)$ .

## Problem G. Game of Quantum Computing Chickens

We first characterize the winning positions of the game. In each move, the current player must remove stones from one of the largest piles.

Let the current multiset have maximum value  $x$ . If  $x$  appears exactly once, let  $y < x$  be the second largest value. The current player may reduce the pile of size  $x$  to  $y$ , forcing the opponent to play from a position whose maximum is  $y$ . If that position is losing, then the original position is winning. Otherwise, the opponent can reduce  $y$  to some  $z$  and reach a losing position. In that case, the current player could instead directly reduce  $x$  to  $z$ , reaching the same losing position for the opponent. Therefore, whenever the maximum value is unique, the position is winning for the first player.

Now consider the case where the maximum value  $x$  appears  $cnt$  times. Any move must operate on one of these maximum piles, so after each move, the number of piles equal to  $x$  decreases by exactly one. Consequently, the first player has a winning strategy if and only if the maximum value in the multiset appears an odd number of times.

For a query operation, let  $cnt_x$  denote the number of occurrences of  $x$ , and let  $\sigma$  be the total number of elements with value at most  $x$ . The number of ways to select an odd number of occurrences of  $x$  is

$$\sum_{\substack{1 \leq i \leq cnt_x \\ i \text{ odd}}} \binom{cnt_x}{i} = 2^{cnt_x-1}.$$

All elements smaller than  $x$  can be chosen arbitrarily, contributing a factor of  $2^{\sigma - cnt_x}$ . Hence, the total contribution of value  $x$  is  $2^{\sigma-1}$ . All operations can be supported with a segment tree, thus the problem can be solved in  $O(n \log V)$  time.

## Problem H. Heaviest Convex Set

For every ordered pair of distinct points  $(i, j)$ , we create a directed edge  $i \rightarrow j$  with weight  $w_j$ . Sort all  $O(n^2)$  directed edges by polar angle in  $[0, 2\pi)$ . Ties are broken by decreasing projection of their starting points along the given direction, which ensures that the edges of any valid convex polygon form a subsequence of this order, while collinear configurations are excluded.

For any convex polygon traversed counterclockwise, its edges satisfy the following properties: the edge with minimum polar angle is  $P \rightarrow Q$ , where  $P$  is the vertex with minimum  $y$ -coordinate (and minimum  $x$  if tied); the polar angles then strictly increase; and the final edge ends at  $P$ .

Since at most  $k$  points may be removed, the optimal polygon must choose  $P$  from the first  $k+1$  points after sorting all points by  $(y, x)$ . Fix such a point  $P$ . The problem then reduces to selecting a subsequence of the polar-angle-sorted edges that starts and ends at  $P$ , uses at least  $n - k$  edges, and maximizes the total weight.

Let  $f[i][j]$  denote the maximum total weight of a path that starts at  $P$ , ends at point  $i$ , and uses exactly  $j$  edges. Processing directed edges  $U \rightarrow V$  in increasing polar angle order, we update  $f[V][j+1] = \max(f[V][j+1], f[U][j] + w_V)$ . For all  $j \geq n - k$ , the value  $f[P][j]$  yields a valid candidate answer.

To reduce complexity, for each  $f[i][\cdot]$  we keep only the largest  $k+1$  states in the second dimension, which suffices for optimality. These transitions can be implemented using two pointers. Moreover, after fixing  $P$ , we sort all other points by polar angle around  $P$ . Any edge whose endpoints differ by more than  $k+1$  in this order cannot appear in a valid solution and is discarded, reducing the number of edges to  $O(nk)$ .

The overall time complexity is  $O(nk^3 + nk^2 \log n)$ .

## Problem I. Ice Skating

A stop occurs only at boundary cells or at cells immediately to the left or above an obstacle. We call such cells key positions. The total number of key positions is  $O(n + m + q)$ . From any key position, the skater has at most one valid outgoing move, and the next stop is again a key position. Therefore, all key positions form a forest that can be maintained using a Link-Cut Tree (LCT). By maintaining the depth of each node, queries can be answered efficiently.

Directly maintaining this forest while adding obstacles may cause large structural changes. Instead, we process all operations in reverse order and handle obstacle removals. Removing an obstacle may introduce one new key position (the obstacle cell itself) and invalidate up to two existing key positions (its left and upper neighbors).

We always introduce key positions as usual. However, it would be hard to explicitly delete invalid key positions. Instead, we eliminate their effect on the answer by modifying the structure locally. Specifically, we split each key position into two nodes: one representing paths that arrive from below, and the other representing paths that arrive from the left. When a key position becomes invalid, the path arriving from below continues upward to find a new parent, and the path arriving from the left continues rightward to find a new parent. This avoids explicit deletion and keeps changes to the forest localized.

The parent of a node, represented by a pair of a position and an incoming direction, can be determined using binary search over ordered sets. The overall time complexity is  $O(q \log(n + m + q))$ .

## Problem J. Jamming of Intelligence

**Preliminary analysis.** Type A and type B intelligence tend to make each submarine's firing range as large as possible, while type C and type D intelligence tend to restrict the firing ranges. We first consider type C and type D intelligence. Each type D constraint  $r_u \leq w$  is a mandatory unary constraint. Each type C constraint  $\min(r_u, r_v) \leq w$  requires that at least one of  $r_u \leq w$  or  $r_v \leq w$  must hold; we choose one to enforce, while the other is optional. If it is allowed to enumerate all choices for type C constraints in  $\mathcal{O}(2^c)$  time, we can obtain an upper bound for each submarine's firing range under the mandatory unary constraints. Assigning each firing range to its upper bound is the best we can do to satisfy type A and type B intelligence.

Thus, deciding whether a feasible assignment exists reduces to a binary constraint selection problem. Thus, we would like to model it as a 2-SAT problem.

**Construct the graph.** For each Boolean variable  $\mathcal{X}$ , let  $x$  denote the node where  $\mathcal{X}$  is true, and  $x'$  denote the node where  $\mathcal{X}$  is false.

1. For each type D intelligence  $(u, w, p)$ , we introduce a Boolean variable  $\mathcal{X}$  meaning "whether  $r_u$  must not exceed  $w$ ". We add an edge  $x' \rightarrow x$  with weight  $p$ , indicating that we may pay cost  $p$  to force this constraint to hold.
2. For each type C intelligence  $\min(r_u, r_v) \leq w$ , we introduce two Boolean variables  $\mathcal{X}$  and  $\mathcal{Y}$  meaning "whether  $r_u$  must not exceed  $w$ " and "whether  $r_v$  must not exceed  $w$ ", respectively. We add zero-cost edges  $x' \rightarrow y$  and  $y' \rightarrow x$ , enforcing that at least one of the two unary constraints must be satisfied.
3. For each submarine, we collect all its associated constraints (variables) and sort them by the corresponding bound in increasing order. For two adjacent variables  $\mathcal{X}$  and  $\mathcal{Y}$  representing " $r_u \leq p$ " and " $r_u \leq q$ " with  $p < q$ , if  $\mathcal{X}$  is true then  $\mathcal{Y}$  must also be true; if  $\mathcal{Y}$  is false then  $\mathcal{X}$  must be false. We add zero-cost edges  $x \rightarrow y$  and  $y' \rightarrow x'$ .
4. For each type B intelligence  $\max(r_u, r_v) \geq w$ , we find the largest variable of  $u$  and  $v$  whose bound is smaller than  $w$ , denoted by  $\mathcal{X}$  and  $\mathcal{Y}$ . These two variables cannot both be true, so we add zero-cost edges  $x \rightarrow y'$  and  $y \rightarrow x'$ . No additional constraints are needed, as transitivity is already guaranteed by step 3.

5. For each type A intelligence  $r_u + r_v \geq w$ , we enumerate each variable  $\mathcal{X}$  of  $u$ , representing " $r_u \leq p$ ", and find the largest variable  $\mathcal{Y}$  of  $v$  whose bound is smaller than  $w - p$ . These two variables cannot both be true, so we add zero-cost edges  $x \rightarrow y'$  and  $y \rightarrow x'$ . Similarly, no further constraints are required based on step 3.

After deduplication, there are at most  $m = 2c + d$  variables. Intelligence of types B, C, and D contributes  $\mathcal{O}(b + c + d)$  edges, but type A intelligence may introduce up to  $\mathcal{O}(a(c + d))$  edges, which is unacceptable.

**Optimizing edge construction for type A.** Let  $\text{size}_i$  denote the number of variables associated with submarine  $i$ , so that  $\sum \text{size}_i = m$ . For each type A intelligence, we may assume  $\text{size}_u \leq \text{size}_v$ ; otherwise, we simply swap  $u$  and  $v$ . We consider the following cases.

- If  $\text{size}_u \leq \text{size}_v \leq k$ , we can apply a two-pointer scan over the variable sets of  $u$  and  $v$  in  $\mathcal{O}(k)$  time, adding  $\mathcal{O}(\text{size}_u)$  edges.
- If  $\text{size}_v > k$ , we preprocess in  $\mathcal{O}(n)$  time an array  $f_{v,i}$ , which denotes the largest variable of  $v$  whose bound does not exceed  $i$ . Then, for each variable of  $u$ , we can locate the corresponding position in  $v$  in  $\mathcal{O}(1)$  time using  $f$ , and add  $\mathcal{O}(\text{size}_u)$  edges in total.

For a fixed submarine  $v$ :

- If  $\text{size}_v \leq \sqrt{m}$ , then each type A intelligence involving  $v$  contributes at most  $\mathcal{O}(\sqrt{m})$  edges.
- If  $\text{size}_v > \sqrt{m}$ , then there are at most  $\mathcal{O}(\sqrt{m})$  such submarines. Moreover, since the corresponding  $u$  are all distinct for a fixed  $v$ , the total number of edges contributed by this  $v$  is at most  $\mathcal{O}(\sum_{i=1}^n \text{size}_i) = \mathcal{O}(m)$ .

Overall, the total number of edges is bounded by  $\mathcal{O}(b + (a + c + d)\sqrt{c + d})$ , which is acceptable.

We now analyze the time complexity of handling type A intelligence. The total cost is  $\mathcal{O}(ak + \frac{nm}{k} + m\sqrt{m})$ , which is minimized when  $k = \sqrt{\frac{nm}{a}}$ , yielding an overall complexity of  $\mathcal{O}(\sqrt{nma} + m\sqrt{m})$ .

**Finding the minimum cost.** We first remove all edges with positive cost and compute the strongly connected components of the graph. If there exists a variable  $\mathcal{X}$  such that  $x$  and  $x'$  lie in the same strongly connected component, then the constraint system is already inconsistent, and no injection is needed.

Otherwise, we enumerate the injected type D intelligence. Suppose a type D intelligence corresponds to an edge  $x' \rightarrow x$ ; in any optimal solution, after injection,  $x$  and  $x'$  must belong to the same strongly connected component.

We construct a directed graph  $\mathcal{G}$  with  $d$  vertices, where each vertex represents a type D intelligence. If, in the original graph, the true node of type D intelligence  $i$  can reach the false node of type D intelligence  $j$  via zero-cost edges (directly or indirectly), then we add an edge  $i \rightarrow j$  in  $\mathcal{G}$  with weight equal to the cost of intelligence  $i$ . The answer is the minimum-weight cycle in  $\mathcal{G}$ . Although the minimum cycle can be found using the Floyd algorithm in  $\mathcal{O}(d^3)$  time, due to the special structure of the 2-SAT graph in this problem, any minimum cycle contains at most two edges, allowing a brute-force solution in  $\mathcal{O}(d^2)$  time.

To construct  $\mathcal{G}$ , we preprocess which false nodes of type D intelligence are reachable from each node in the 2-SAT graph, using bitsets for acceleration.

The overall time complexity is  $\mathcal{O}\left(\sqrt{na(c + d)} + \frac{d(b + (a + c + d)\sqrt{c + d})}{w} + d^2\right)$ .

## Problem K. Keeping Cactus Alive

First, the overall running time is bounded by the minimum durability among all bridge edges. Every non-bridge edge lies in exactly one simple cycle. Consider a cycle of length  $\ell$ . At any time, at least  $\ell - 1$

edges on the cycle must remain active to keep the graph connected. Let the durabilities of the edges on this cycle be sorted as  $w_1 \leq w_2 \leq w_3 \leq \dots$ . Then the maximum sustainable time for this cycle is  $\min(w_1 + w_2, w_3)$ . In particular, for a cycle of length 2, this reduces to  $w_1 + w_2$ .

For implementation, it is unnecessary to explicitly compute bridges or BCCs since the graph is guaranteed to be a cactus. We first construct an arbitrary spanning tree, then, for each non-tree edge, traverse the unique path in the tree between its endpoints to identify the corresponding cycle. Edges that are not covered by any such traversal are bridges. This procedure runs in linear time, as each edge belongs to at most one cycle.

## Problem L. Love Choice

Let the random variable  $x_i$  denote the round in which problem  $i$  is guessed correctly for the first time, and let  $X = \max_{1 \leq i \leq n} x_i$  be the first round in which all problems are answered correctly.

We have  $X \in [L, R]$ , where  $L = \max_{1 \leq j \leq m} (a_{b_j} + 1)$  and  $R = \max(L, \max_{1 \leq i \leq n} a_i)$ . Since  $X$  is a positive integer-valued random variable,

$$\mathbb{E}[X] = L + \sum_{k=L+1}^R \Pr(X \geq k).$$

Moreover,

$$\Pr(X \geq k) = 1 - \Pr(X < k) = 1 - \prod_{i=1}^n \Pr(x_i < k).$$

Therefore,

$$\begin{aligned} \mathbb{E}[X] &= L + \sum_{k=L+1}^R \left( 1 - \prod_{i=1}^n \Pr(x_i < k) \right) \\ &= R - \sum_{k=L}^{R-1} \prod_{i=1}^n \Pr(x_i \leq k). \end{aligned}$$

For problems with hidden answers, we have  $x_i = a_i + 1 \leq L \leq k$ , so  $\Pr(x_i \leq k) = 1$  always holds. For the remaining problems, we examine the distribution of  $x_i$ . For  $1 \leq t \leq a_i$ ,

$$\Pr(x_i = t) = \frac{a_i - 1}{a_i} \cdot \frac{a_i - 2}{a_i - 1} \cdots \frac{a_i - t + 1}{a_i - t + 2} \cdot \frac{1}{a_i - t + 1} = \frac{1}{a_i},$$

which shows that  $x_i$  is uniformly distributed over  $\{1, \dots, a_i\}$ . Hence,

$$\Pr(x_i \leq k) = \begin{cases} \frac{k}{a_i}, & a_i \geq k, \\ 1, & a_i < k. \end{cases}$$

Substituting into the expectation formula yields

$$\mathbb{E}[X] = R - \sum_{L \leq k \leq R-1} \prod_{a_i \geq k} \frac{k}{a_i} = R - \sum_{L \leq k \leq R-1} \left( \prod_{a_i \geq k} \frac{1}{a_i} \right) k^{\text{cnt}(k)},$$

where

$$\text{cnt}(k) = \sum_{1 \leq i \leq n} [a_i \geq k].$$

Observe that  $\text{cnt}(k)$  is non-increasing in  $k$ , and all values with the same  $\text{cnt}(k)$  form a continuous interval. Moreover, the coefficient  $\prod_{a_i \geq k} \frac{1}{a_i}$  is constant over such an interval. Therefore, each segment reduces to computing a sum of powers of consecutive integers, which can be handled using Lagrange interpolation or other standard techniques.

The overall time complexity is  $O(n^2)$ .

## Problem M. Metronome for Corgi

Given a sequence, find the minimum  $k$  such that the sequence can be partitioned into several segments, where the sum of the last segment is at most  $k$ , and the sum of every other segment is exactly  $k$ .

Let  $s_i = \sum_{j=1}^i a_j$  denote the prefix sums of the sequence. Considering the first segment, any feasible  $k$  must be equal to some prefix sum  $s_i$ . Therefore, all candidates for  $k$  can be enumerated in  $O(n)$  time.

We first insert all prefix sums into a data structure that supports fast membership queries. For a fixed candidate  $k$ , we check, in increasing order, whether  $k, 2k, 3k, \dots$  appear among the prefix sums, stopping as soon as a multiple is missing. The candidate  $k$  is feasible if and only if all such multiples up to the total sum exist, except possibly the final remainder.

We call a subset of  $\mathbb{N}$  a homogeneous arithmetic progression if it is of the form  $\{a, 2a, 3a, \dots, ba\}$  for some integers  $a \geq 1$  and  $b \geq 1$ . For a fixed  $k$ , the successful multiples  $k, 2k, \dots, bk$  that appear in the prefix sum set form exactly such a homogeneous arithmetic progression with difference  $k$ .

Consequently, the total number of successful multiple checks performed by the algorithm is equal to the total number of homogeneous arithmetic progressions contained in the set of prefix sums. For a finite set  $X \subseteq \mathbb{N}$  with  $|X| = n$ , the best known upper bound on the number of homogeneous arithmetic progressions in  $X$  is  $(1 + o(1))n \log^2 n$ ; see [https://glaudo.github.io/files/hap\\_bound.pdf](https://glaudo.github.io/files/hap_bound.pdf) for a proof.

Thus, our algorithm runs in  $O(n \log^2 n)$  time.