Problem A. Acridity and Sourness

Problem Outline

- Given the acridity a_i and sourness s_i of n types of condiments.
- For all possible pairs of condiments, determine whether the target acridity x and sourness y can be obtained by mixing them in an appropriate ratio.

Constraints

- $n \leq 50$
- $0 \leq a_i, s_i, x, y \leq 50$
- $(a_i, s_i) \neq (x, y)$
- All inputs are integers.

Solution

- Plot the acridity and sourness values on a two-dimensional plane. The flavor coordinates that can be produced by mixing condiments i and j lie strictly on the line segment connecting (a_i, s_i) and (a_j, s_j).
- Whether the target point (x, y) lies on this line segment can be determined in $O(1)$ constant time using vector operations such as dot products and cross products.
- Alternatively, if a valid mixture exists, there is a mixing ratio $p : q$ where p and q are positive integers satisfying $p + q \leq 50$. Thus, you can also solve the problem by exhaustively searching all possible ratios.

Problem B. Broken Key

Solution (DP Approach)

- Let $\mathrm{dp}(i, c)$ be the minimum number of times **Operation 1** is performed such that the i-th character of S has been appended, and the character c is currently not contained in X.
- A naive transition takes $O(|S|\sigma^2)$ time, where $\sigma = 26$ is the alphabet size.
- By limiting the transitions to only replacing the most recently added character with another character, the transition complexity can be reduced to $O(|S|\sigma)$ time.

Solution ($O(|S|)$ Approach)

- There is no need to perform **Operation 1** at all before the first occurrence of `'z'` in S.
- After that, each time we perform **Operation 1** once, we can continue performing **Operation 2** as long as the number of distinct characters in the current substring is at most 25. This greedy strategy is optimal.
- Overall time complexity: $O(|S|)$.

Problem C. Counting Triangles

Solution (Triangulation Properties)

- If all n points lie on a single straight line, the number of triangles is 0.
- Otherwise, when there are n total points and m points lie on the boundary of their convex hull, any triangulation of these points will yield exactly $2n - m - 2$ triangles.
- $\rightarrow$ Consequently, to maximize the number of triangles, we should minimize the number of points m that lie on the boundary of the new convex hull.

Case $k \geq 2$

- When $k \geq 2$, we can always achieve $m = 3$ on the new convex hull. This is done by placing two of the newly added points far enough away to form a giant triangle that encloses all other $N + k - 3$ points.
- $\rightarrow$ Thus, the sum of the answers for all $k = 2, \ldots, K$ can be computed in $O(1)$ time using the formula $2(N + k) - 3 - 2 = 2N + 2k - 5$.

Case $k = 1$

- When $k = 1$, let the points on the convex hull of the original N points be labeled $1, 2, \ldots, m$ in counter-clockwise order.
- By adding exactly one point, can we construct a new convex hull such that only a contiguous range of original hull points $i, i + 1, \ldots, j$ (index arithmetic modulo m) remain on the boundary of the new hull?
- This feasibility check can be performed by verifying the intersection of the line passing through points i and $i - 1$ and the line passing through points j and $j + 1$.
- We can find the minimum value of $j - i$ using a two-pointer approach (or binary search).

Problem D. Do the Inspection

Solution

- Compute the XOR sum of all edges of the rooms (segments that appear an odd number of times in the input rectangles).
  - For all horizontal edges, it is efficient to compute the prefix sums from the left for each y-coordinate. The same logic applies to the vertical edges.
- Constructing an undirected graph from the remaining boundary segments decomposes the boundary into several disjoint cycles. Exactly one of these cycles represents the outer boundary of the house, and the rest represent the strange spaces. Thus, we can find the strange spaces by simply excluding the cycle with the maximum area.
- The overall algorithm can be implemented to run in $O(n \log n)$ time.
- Alternative approaches include using Euler's polyhedron formula to count the number of regions and tracing boundary cycles (e.g., using the left-hand rule) to compute areas.

Problem E. Emerald Shop

Solution

- Consider a graph where items are vertices and people are edges. If we find the answer for each connected component, we can multiply these answers together and multiply by an appropriate multinomial coefficient to find the total answer.
- For a connected component to have a valid solution, it must be either a tree or a pseudo-tree (a component with exactly one cycle).
- We will find the answer for each case.
- Assume a directed edge is drawn from A_i to B_i.

The Case of a Tree

- The out-degree of the single unsold vertex in the component must be 0 (i.e., it cannot appear in A).
- Fix the unsold vertex and treat it as the root. For each vertex i, let S_i be the following value:
  - The number of descendants of vertex i that can be reached from i.
- Then, the answer is $N!$ multiplied by the product of the reciprocals of S_i.
- S_i can be computed using tree DP, or rerooting tree DP (all-directions tree DP) to handle all possible roots.

The Case of a Pseudo-tree

- Looking at the cycle part, there are only 2 possible ways of purchasing, so we fix the orientation of the cycle and consider them.
- Similar to the tree case, the product of the reciprocals of S_i directly gives the answer.
- Note that we treat the cycle itself as the root. Care is needed when all edges in the cycle point in the same direction.
- The implementation is quite challenging.

Problem F. Function and String

Solution (1)

- Let us consider the number of required operations for each substring.
- If the string can be transformed into "01", the number of operations is $|T| - 2 +$ (number of swaps).
- If the string can be transformed into "0" or "1", the number of operations is $|T| - 1 +$ (number of swaps).
- Therefore, it is sufficient to determine:
  - Whether each substring can be transformed into "0", "1", or "01", and
  - The minimum number of swaps required to achieve that transformation.

Solution (2)

- Consider the replacements:
  - **Operation 2:** Choose "00" and replace it with "1".
  - **Operation 3:** Choose "11" and replace it with "0".
- We can map the characters to elements in $\mathbb{Z}_3$ (integers modulo 3) as follows:
  - **Operation 2:** Choose "11" and replace it with "2".
  - **Operation 3:** Choose "22" and replace it with "1".
- This corresponds to replacing two adjacent identical values with their sum modulo 3, which is an invariant (the sum of the array elements modulo 3 remains constant).
- Using DP or similar techniques, we can count the number of substrings that can be successfully transformed into "0", "1", or "01".

Solution (3)

- Swaps (Operation 1) are necessary only when '0' and '1' appear alternately in the string (excluding the target strings "0", "1", and "01" themselves).
- If the string does not consist of alternating characters, no swaps are needed:
  - Generally, choosing a "00" adjacent to a '1' (or "11" adjacent to a '0'), such as $\ldots 0001 \cdots \to \ldots 0011 \ldots$, allows the subsequent Operation 2 or 3 to be performed (or terminates the process).
  - Although it seems we might reach the state "10", the only states immediately preceding "10" are "01", "000", and "111". The latter two ("000" and "111") can be transformed into "01" with exactly one operation.
- By flipping the characters at odd indices and applying run-length compression, the total number of swaps can be easily computed.

Problem G. Giant Playlist

Solution

- Let $t[i] + x[i]$ seconds be the time person i gets excited if we start the playlist from the beginning of song 1.
- If we start playing the playlist from a offset of y seconds (where y is the starting position relative to song 1), the time person i gets excited is:
  - If $y \leq x[i]$: $t[i] + x[i] - y$ seconds
  - If $y > x[i]$: $t[i] + x[i] - y + \sum L$ seconds
- If we sort the M people in ascending order of $x[i]$ and fix the song to start with (which determines the value of y), the people are split into the two cases above at some boundary. The minimum time to satisfy everyone for each starting song can then be calculated using prefix/suffix maximums (cumulative max) or a segment tree.

Problem H. How Many Ways to Pack?

Solution

- Assume $A[1] \leq A[2] \leq \cdots \leq A[N]$. Let us determine if $S = \{1, 2, \ldots, N\}$ is good for a given w.
- Any ball heavier than $w - A[1]$ must form its own group consisting of only that single ball.
- If the sum of the weights of all balls with weight at most $w - A[1]$ is at most w, they must be grouped together into a single group.
- What if that is not the case?
- If the sum of the weights of the balls with weight at most $w - A[1]$ is strictly greater than w:
  - For each i such that $A[i] \leq w - A[1]$, there exists a valid partitioning where ball 1 and ball i belong to the same group.
  - However, it is impossible to put all balls with weight at most $w - A[1]$ into a single group.
  - $\rightarrow$ This implies there are at least two different valid partitionings!
- Assuming $A[1] \leq A[2] \leq \cdots \leq A[N]$, the necessary and sufficient condition for $S = \{1, 2, \ldots, N\}$ to be good with respect to w is:
  1. $A[N] \leq w$
  2. The sum of the weights of the balls in S whose weight is at most $w - A[1]$ is at most w.
- We can perform the counting by sorting the balls in descending order of their weights and using dynamic programming (DP).

Problem I. In One Row

Ordering Blocks

For any fixed consecutive block of vertices, store three values:

$$\text{info}(B) = (z_B, o_B, c_B),$$

where z_B is the number of zeros in the block, o_B is the number of ones in the block, and c_B is the number of inversions completely inside this block.

If two blocks A and B are put in this order, the number of inversions contributed between them is $o_A z_B$. Therefore, putting A before B is no worse than putting B before A if and only if

$$o_A z_B \le o_B z_A.$$

Thus blocks should be sorted by increasing ratio $\frac{o}{z}$. We never need to divide: compare two blocks by cross multiplication. Define

$$A \prec B \quad \Longleftrightarrow \quad o_A z_B < o_B z_A.$$

Ties may be broken arbitrarily, because both orders give the same cross contribution.

If a block A must be immediately followed by a block B, we replace them by one block

$$\text{merge}(A, B) = (z_A + z_B,\ o_A + o_B,\ c_A + c_B + o_A z_B).$$

Solving One Tree

Consider one rooted tree. We will transform it into a list of blocks. Later, these blocks can be sorted by $\prec$, and their concatenation is an optimal valid order for this tree.

Process vertices bottom-up. For every vertex u, maintain a priority queue of blocks belonging to already processed child subtrees of u, ordered by $\prec$. To process u:

1. Merge all priority queues of the children of u into the priority queue of u.
2. Create the current block C consisting only of u:

$$C = (1, 0, 0) \text{ if } v_u = 0, \qquad C = (0, 1, 0) \text{ if } v_u = 1.$$

3. While the smallest block B in the priority queue satisfies $B \prec C$, merge it into C:

$$C \leftarrow \text{merge}(C, B).$$

   Then remove B from the priority queue.

4. Insert the final block C into the priority queue of u.

Why is this correct? Suppose the smallest available child block is B and $B \prec C$. If there were no tree constraint, B should be placed before C. However, every vertex in B is a descendant of u, while C contains u, so B cannot be placed before C. Among all blocks that must be placed after C, choosing the smallest one next is optimal by the adjacent-swap argument above. Therefore B is forced to be immediately after C, and the two blocks can be merged. We repeat this until no remaining child block wants to be before C.

After processing the root, its priority queue contains all final blocks of the tree in sorted order. If these blocks are $B_1, B_2, \ldots, B_s$, the minimum inversion count inside this tree is

$$\sum_{i=1}^{s} c_{B_i} + \sum_{1 \le i < j \le s} o_{B_i} z_{B_j}.$$

The second sum can be computed by scanning the blocks from left to right and maintaining the prefix sum of ones.

Combining Two Trees

For Ryan's tree, compute its final sorted block list once. Let it be

$$R_1, R_2, \ldots, R_s.$$

Also precompute:

- the optimal inversion count inside Ryan's tree, denoted by base_R;
- prefix sums of zeros and ones over the list R.

For each friend's tree, after decrypting its labels, compute its final sorted block list

$$G_1, G_2, \ldots, G_t$$

and its internal optimal inversion count base_G in the same way.

There are no ancestor constraints between the two trees. Therefore, after both trees have been reduced to sorted blocks, the optimal combined order is just the sorted merge of the two block lists using the same order $\prec$.

We do not need to actually merge the two lists. For a block G from the friend's tree, let $pos(G)$ be the number of Ryan blocks strictly smaller than G. Then:

- Ryan blocks before G contribute

$$z_G \cdot \left(\text{number of ones in } R_1, \ldots, R_{pos(G)}\right);$$

- Ryan blocks after G contribute

$$o_G \cdot \left(\text{number of zeros in } R_{pos(G)+1}, \ldots, R_s\right).$$

Since the Ryan blocks are sorted, $pos(G)$ is found by binary search. Summing this over all blocks of the friend's tree gives the cross inversions $\text{cross}(R, G)$, and the answer for this friend is

$$\text{base}_R + \text{base}_G + \text{cross}(R, G).$$

Complexity

For a tree with m vertices, the priority queues can be merged small-to-large. Each block is moved $O(\log m)$ times, and each priority queue operation costs $O(\log m)$, so this implementation runs in

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

time.

For each query, after reducing the friend's tree, each of its final blocks performs one binary search in Ryan's block list. Therefore, the total additional cost over all queries is

$$O\left(\left(\sum m_k\right) \log n\right).$$

Problem J. Japanese Art

- For any two points to be connected, there are exactly two possible polygonal chains depending on their relative positions.
- Assign a boolean variable to each color, representing one of the two choices as True and the other as False.
  - For example, if one point is at the bottom-left of the other, the two possible chains are ⌊ and ⌉.
- Check for intersections between all pairs of possible chains and reduce the problem to 2-SAT.
  - For example, if choosing chain x and chain y simultaneously is impossible because they intersect, we can represent this constraint as a 2-SAT clause $\neg(x \wedge y) \equiv \neg x \vee \neg y$.
- Since there are $O(n^2)$ pairs of chains, the total number of clauses is $O(n^2)$. Thus, the problem can be solved in $O(n^2)$ time.

Problem K. Know Your Prefix

- If the length of A is less than M, we can simulate the operations naively (this happens at most $O(\log M)$ times).
- Once the length of A exceeds M, we can discard everything after the first M terms. Thus, the operation can be rephrased as follows:
  - **Operation:** Replace the sequence A with $(A[1], A[2], \ldots, A[x], A[1], A[2], \ldots, A[M-x])$.

Operation: Replace A with $(A[1], A[2], \ldots, A[x], A[1], A[2], \ldots, A[M-x])$.

- The prefix of the shorter sequence between $(A[1], \ldots, A[x])$ and $(A[1], \ldots, A[M-x])$ matches the prefix of the longer sequence.
- Therefore, we only need to know the first $\max(x, M-x)$ terms of A after the previous operation.
- By analyzing the queries in reverse order, we can determine how many prefix elements are required at each step and then construct the sequence from the beginning.
  - Implementation details: e.g., filling the array in-place, using a `std::deque`, etc.

Problem L. Leave Only Root

1 Problem Formulation and Recurrence Relations

Let $T = (V, E)$ be a tree rooted at vertex 1. We are allowed to repeatedly delete a leaf. If a deleted leaf u is a child of the root (vertex 1), nothing happens. Otherwise, let p be the parent of u and gp be the parent of p. The deletion of u causes K new copies of the subtree rooted at p (excluding u) to be attached to gp.

Our goal is to find the minimum number of operations required to reduce the tree to only vertex 1.

1.1 Effective Weight of a Subtree

Let $E(u)$ denote the **effective weight** of the subtree rooted at u. This weight represents the multiplicative impact of the subtree when its parent is processed.

- If u is a leaf, deleting it does not trigger any replication of its siblings. Thus, its base contribution is:
$$E(u) = 1$$
- If u is an internal node with a set of children $\mathcal{C}(u) = \{v_1, v_2, \ldots, v_d\}$, deleting the leaves of its children recursively triggers replication. Specifically, deleting a component of v_i multiplies the remaining structure by $K + 1$ copies. By induction, the effective weight of u satisfies:
$$E(u) = 1 + \sum_{v \in \mathcal{C}(u)} (K+1)^{E(v)-1}$$

1.2 Cost to Resolve a Subtree

Let $dp[u]$ be the minimum number of operations to completely delete the subtree rooted at u, assuming u's parent is not deleted. Let $R[u]$ represent the accumulated cost of processing all descendants of u up to the point where only u remains.

- For a leaf node u:
$$R[u] = 0$$
- For an internal node u:
$$R[u] = \sum_{v \in \mathcal{C}(u)} (R[v] + h(E(v) - 1)) \pmod{998244353}$$
where $h(x)$ is the cost of collapsing an effective weight of x, defined as:
$$h(x) = \frac{(K+1)^x - 1}{K}$$

The total cost to resolve the subtree at u (when its parent is intact) is:

$$dp[u] = h(E(u)) + R[u] \pmod{998244353}$$

For the root (vertex 1), its children can be deleted independently without triggering parent-level replication. Thus, the total minimum operations is:

$$\text{Total Cost} = \sum_{u \in \mathcal{C}(1)} dp[u] \pmod{998244353}$$

2 Managing Tower Exponentials via Euler's Totient Chain

The definition of $E(u)$ contains tower exponentials of the form $(K+1)^{(K+1)^{\cdots}}$. These values grow extremely quickly and cannot be stored in standard computer word sizes. We must compute these values modulo $M_0 = 998244353$.

2.1 The Modulo Chain

According to the **Generalized Euler's Totient Theorem**, for any integers a, x and modulus m:

$$a^x \equiv a^{(x \bmod \phi(m))+\phi(m)} \pmod{m} \quad \text{for } x \geq \phi(m)$$

To compute $E(u) \pmod{M_0}$, we must compute the exponent $E(v) - 1$ modulo $\phi(M_0)$. This requires computing the exponent of *that* term modulo $\phi(\phi(M_0))$, and so on. We define a descending chain of moduli:

$$M_0 = 998244353$$
$$M_{i+1} = \phi(M_i) \quad \text{for } i \geq 0$$

Because $\phi(m)$ is even for all $m > 2$, the value of M_i halves at least every two steps. Thus, the chain terminates at 1 very quickly. The chain has a length $L = 31$.

Mathematical Proof of Modulo Chain Finiteness

Let m be an even integer. The Euler totient function $\phi(m)$ counts the integers $1 \leq k \leq m$ coprime to m. Since m is even, any coprime k must be odd. Hence, $\phi(m) \leq \frac{m}{2}$. For any odd integer m, $\phi(m)$ is even. Thus, after at most one step, the modulus becomes even, and subsequent steps halve the value at least every two iterations:

$$\phi(\phi(m)) \leq \frac{m}{2}$$

This guarantees logarithmic decay, limiting the chain length to $L \leq 2\log_2(M_0) \approx 60$. For 998244353, it terminates in exactly 31 steps.

3 Transition System and "Large" Value Tracking

Because Euler's theorem behaves differently depending on whether the exponent x is less than or greater than $\phi(m)$, we must track if $E(u)$ is large. We define a threshold $T_{\text{large}} = 10^9 + 7$. Since $T_{\text{large}} > M_i$ for all $i \geq 0$, any value $E(u) \geq T_{\text{large}}$ is guaranteed to be larger than $\phi(M_i)$ for any modulus in our chain.

3.1 DP State Representation

For each node u, we store:

1. `large_val[u]`: A boolean flag indicating if $E(u) \geq T_{\text{large}}$.
2. `exact_E[u]`: The exact value of $E(u)$ if `large_val[u]` is false.
3. `E[u][i]`: The value of $E(u) \pmod{M_i}$ for $0 \leq i \leq L$.

3.2 State Transitions

For a node u with children $\mathcal{C}(u)$:

1. Exact Value Estimation: To avoid integer overflow, we compute:

$$\text{exact\_E}[u] = 1 + \sum_{v \in \mathcal{C}(u)} (K+1)^{\text{exact\_E}[v]-1}$$

If at any point during this summation $\text{exact\_E}[u] \geq T_{\text{large}}$ or any child has `large_val[v] == true`, we set `large_val[u] = true`.

2. Modular Exponentiation: For each modulus index $i \in [0, L]$:

$$E[u][i] = \left(1 + \sum_{v \in \mathcal{C}(u)} (K+1)^{\exp(v,i)}\right) \pmod{M_i}$$

where the exponent term $\exp(v, i)$ is determined by the scale of v:

- If `large_val[v]` is false and $\text{exact\_E}[v] - 1 < M_{i+1}$:

$$\exp(v, i) = \text{exact\_E}[v] - 1$$

- Otherwise (the exponent is large relative to the modulus $\phi(M_i) = M_{i+1}$):

$$\exp(v, i) = ((E[v][i+1] - 1) \bmod M_{i+1}) + M_{i+1}$$

4 Complexity Analysis

4.1 Time Complexity

- **Modulus Chain Generation:** Finding $\phi(M_i)$ takes $O(\sqrt{M_i})$ time. Since M_i decreases exponentially, this step takes $O(\sqrt{M_0}) \approx 31594$ operations, which is negligible.
- **Topological Ordering:** A BFS traversal takes $O(N)$ time.
- **DP Transitions:** For each node u, we iterate over its children $\mathcal{C}(u)$ and compute powers for L moduli. Using binary exponentiation, calculating $(K+1)^{\exp} \pmod{M_i}$ takes $O(\log M_i)$ time. Thus, the total transition cost for node u is $O(|\mathcal{C}(u)| \cdot L \cdot \log M_0)$. Summing over all nodes V:

$$\sum_{u \in V} O(|\mathcal{C}(u)| \cdot L \cdot \log M_0) = O(N \cdot L \log M_0)$$

4.2 Space Complexity

The adjacency list takes $O(N)$ space. The state arrays `E[N][L]` require storing 31 integers per node. Thus, the total space complexity is $O(N \cdot L)$.