Problem A. A Calibration Lab

Problem Overview

- There are N pairs of integers $(A[i], B[i])$.
- Count the number of ordered pairs of two different indices (i, j) such that the four integers $\{A[i], B[i], A[j], B[j]\}$ can be partitioned into two identical multisets.

Solution

- First, ensure $A[i] \leq B[i]$ for each i.
- The required ordered pairs fall into two types:
  1. Both pairs consist of the same integer: (a, a) and (b, b).
  2. The two pairs are equal: (a, b) and (a, b), where $a \neq b$.
- For type 1, maintain a counter for pairs with identical elements.
- For type 2, use a map or similar structure to count occurrences of each distinct pair (a, b) with $a < b$.

Problem B. Box

Problem Overview

- Rearrange a sequence A of N non-negative integers.
- For each index i, add $\left\lfloor \frac{\sum_{j<i} A[j]}{A[i]} \right\rfloor$ to the total cost.
- Find the minimum possible total cost.

Solution

- Sort A in non-decreasing order.
- **Proof (Exchange Argument):** Suppose $A[i] > A[i+1]$. Swapping these two elements affects the cost as follows:
  - For $A[i]$: The numerator increases by $A[i+1]$, so the change in its term is $\leq +1$.
  - For $A[i+1]$: The numerator decreases by $A[i]$, so the change in its term is ≤ -1.

  Thus, the total change in cost is ≤ 0, meaning the swap does not increase the cost.
- Therefore, the optimal arrangement is the sorted order.

Problem C. Chairs and Columns

Problem Overview

- Given an $H \times W$ grid. Some cells are selectable, some are not.
- Cells that are vertically or horizontally adjacent cannot be selected simultaneously.
- Goal: Select as many cells as possible (maximum independent set).
- For each cell, determine if it can be included in some maximum independent set.
- Constraints: $H \leq 400$, $W \leq 400$.

Key Known Fact

A grid graph is bipartite. (Proof: Color like a checkerboard.) Thus, it suffices to solve the following problem: Given a bipartite graph $G = (L, R; E)$, determine for each vertex whether it can be part of some maximum independent set. Let $V = L \cup R$.

Important Fact for Bipartite Graphs

Let M be the size of a maximum matching and I the size of a maximum independent set. Then,

$$M + I = |V|.$$

Brief proof: Show that maximum matching = minimum vertex cover. A minimum vertex cover can be expressed in a "burn/flood-fill" form, which yields the graph for finding a maximum matching.

Observations

For $v \in V$, the following statements are equivalent:

1. There exists a maximum independent set containing v.
2. Removing v and its neighbors $\delta(v)$ from G reduces the size of a maximum independent set by 1.
3. Removing v and its neighbors $\delta(v)$ from G reduces the size of a maximum matching by $|\delta(v)|$.
4. Removing only the neighbors $\delta(v)$ from G reduces the size of a maximum matching by $|\delta(v)|$.

We will consider the last condition.

Matching and Residual Graph

Fix an arbitrary maximum matching M of G. Let $U \subseteq L$ be the set of left vertices unmatched in M. Construct a directed residual graph G' where:

- Edges in M are directed from right to left.
- Other edges are directed from left to right.

Condition for a Left Vertex

For $v \in L$, the following are equivalent:

- Removing v reduces the maximum matching size by 1.
- v is **not** reachable from any vertex in U in G'.

Reasoning: If v is reachable from U, we can swap edges along the alternating path to create a new maximum matching that does not use v, so removing v does not decrease the matching size.

Proof Sketch

Suppose removing v does not decrease the maximum matching size. Show this implies v is reachable from U (for $v \notin U$). Take a new maximum matching M' that does not use v. Consider the symmetric difference of M (red) and M' (blue). Since each vertex has degree at most 2, it consists of simple paths and cycles. By maximality, there is no path starting and ending with blue. Since $|M| = |M'|$, there is also no path starting and ending with red. Because $v \notin U$ and M' does not use v, v is incident only to a red edge. Thus, there exists a simple path ending at v with pattern: blue $\to$ red $\to \cdots \to$ blue $\to$ red. The starting vertex is incident only to a blue edge, so it belongs to U. This path corresponds to a path from U to v in G'.

Implication

If removing $v \in L$ reduces the maximum matching size by 1, then removing v from G' yields a valid residual graph. Removing v does not increase the set of vertices reachable from U.

Conclusion for Right Vertices

For $u \in R$, the following are equivalent:

- There exists a maximum independent set containing u.
- No vertex in $\delta(u)$ (its neighbors) is reachable from U in G'.

Complexity

A maximum bipartite matching can be found in $O(E\sqrt{V})$ time. Performing BFS once gives answers for all right vertices; doing it twice gives answers for all vertices. Total complexity: $O(HW\sqrt{HW})$.

Additional Note (DM Decomposition)

The decomposition of the vertex set obtained by contracting SCCs in the residual graph is called the Dulmage–Mendelsohn (DM) decomposition. It provides various insights about bipartite matchings and can also be used here.

Problem D. Do Count The Inversions

Problem Overview

- Given a string S and an integer K.
- Find the number of inversions in the suffix array of the string formed by repeating S K times.

Solution

1. **Preprocess** S: Ensure S has no period (use Z Algorithm, $O(N)$).
2. **Build Suffix Array for** $S+S$: Construct the suffix array P for the string $S+S$. This can be done with SA-IS in $O(N)$, or using cyclic shifts in $O(N \log N)$.
3. **Compute Inversion Contribution**: Traverse P from left to right while maintaining a Binary Indexed Tree (BIT) to compute contributions to the total inversion count in $O(N \log N)$. For each position i:
   - If $P[i] < N$:

     $$\text{Add } \frac{(K-1)(K-2)}{2} \times \text{BIT.sum}(0, P[i]) \qquad \text{(Between-blocks contribution 1)}$$
     $$\text{Add } \frac{K(K-1)}{2} \times \text{BIT.sum}(P[i]+1, N) \qquad \text{(Between-blocks contribution 2)}$$
     $$\text{Add } (K-1) \times \text{BIT.sum}(N, 2N) \qquad \text{(Block vs } P[i] > N(K-1) \text{ contribution)}$$
     $$\text{Add } \frac{(K-1)(K-2)}{2} \qquad \text{(Within-block contribution)}$$

   - If $P[i] \geq N$:

     $$\text{Add BIT.sum}(P[i]+1, 2N)$$

   Update BIT with $P[i]$ after processing.

Problem E. Earth and Circles

Problem Overview

- There are N non-intersecting circles on a sphere.
- The distance between two points on the sphere is defined as the minimum number of circles crossed when moving between them.
- For a chosen point, consider the maximum distance to all other points.
- Find the minimum possible value of this maximum distance over all points.
- $N \leq 2000$.
- Each circle is given as the intersection of the sphere with a plane defined by its normal vector.

Solution

- Each circle divides the sphere into two regions. Take the smaller region as a "cell".
- The set of these cells (including the whole sphere) forms a laminar family, which induces a tree with $N + 1$ vertices.
- The answer is half the diameter of this tree, rounded up to the nearest integer.

Constructing the Tree

- **Containment relation:** For two cells, check whether a representative point of one lies outside the plane of the other.
- **Parent determination:** For each cell, its parent is the most deeply contained cell among those that contain it.
- If no containing cell exists, its parent is the root (the whole sphere).

Problem F. Find the Minimal Number of Moves

Problem Overview

- A huge binary tree is traversed under the following rules:
- On the i-th move, choose one of:
  - Move to the left child if $S = L$, or to the right child if $S = R$.
  - Move to the parent (only if the parent exists).
- For given nodes u and v, determine if v is reachable from u, and if so, find the minimum number of moves required.
- There are Q queries; u and v can be very large.

Solution

A feasible optimal route follows the path:

$$u \to \mathrm{lca}(u, v) \to v$$

- $u \to \mathrm{lca}(u, v)$: Move upward to the lowest common ancestor as quickly as possible.
- $\mathrm{lca}(u, v) \to v$: Two cases when descending:
  - If the required child direction matches S, move directly.
  - Otherwise, move to the opposite child first, then return upward in the next step.
- Any optimal movement can be transformed into this form without losing optimality.

Implementation

1. **Precomputation:** For each $i = 1, 2, \ldots, N$, compute the first time after i when we can move left/right using a two-pointer method. Note: Only consider positions with the same parity as i.
2. **Query Processing:** Simulate the movement directly. Complexity per query is $O(\log u + \log v)$. Even with naive LCA calculation (repeatedly halving the larger of u and v until they match), the solution passes with careful implementation.

Problem G. Grid And Its Beauty

Problem Overview

- Given a grid with white/black cells.
- Repeatedly perform the following operation:
  - Choose a shortest path from $(1, 1)$ to (H, W) and flip the colors (white $\leftrightarrow$ black) of all cells on that path.
- Goal: Maximize the number of black cells in the final grid.
- Additionally, handle online queries of two types:
  - Flip all cells in a given column.
  - Flip all cells in a given row.
- After each query, output the answer for the current grid.
- Constraints: $H, W, Q \leq 2 \times 10^5$.

Solution

Case: Initially all cells are white

Consider two specific shortest paths:

1. $(1, 1) \to \cdots \to (i, j) \to (i + 1, j) \to (i + 1, j + 1) \to \cdots \to (H, W)$
2. $(1, 1) \to \cdots \to (i, j) \to (i, j + 1) \to (i + 1, j + 1) \to \cdots \to (H, W)$

Applying both flips only cells $(i + 1, j)$ and $(i, j + 1)$ twice, i.e., flips them exactly twice (no net change), while flipping all other cells once. This allows flipping any chosen pair of adjacent diagonal cells.

Greedy construction

By repeatedly applying such paired operations, we can turn the entire $(H-1)\times(W-1)$ top-left rectangular region black. The remaining cells form a path along the last row and last column. The state (color) of each remaining cell is determined by the parity of the number of black cells among cells with the same $i + j$ value.

General case (initial grid may contain black cells)

The effect of initial black cells can be incorporated by flipping the parity condition for remaining cells according to the parity of the total number of initial black cells.

Handling queries

Each row or column flip affects the parity of certain diagonal groups. This reduces to a **range flip + range sum** problem over an array indexed by $i + j$. A lazy segment tree can process each query in $O(\log(H + W))$ time.

Complexity

Total time complexity: $O((H + W + Q)\log(H + W))$.

Problem H. Handle The Queries On Tree

Problem Overview

- Given a tree with N vertices, each vertex has a color.
- Answer Q queries of the form:

  $$(u_1, v_1, u_2, v_2)$$

  Find the largest integer $K \leq N$ such that for every $j = 1, 2, \ldots, K$, the number of vertices of color j on the path $u_1 - v_1$ equals the number of vertices of color j on the path $u_2 - v_2$.

Observations

Special Case: Checking $K = N$

This reduces to checking whether the multiset of colors on the two paths are identical. We can assign a random hash value h_c to each color c. Define the value of a vertex of color c as h_c. Then the two multisets are identical if and only if the sum of vertex values on the two paths are equal (with high probability).

General Case

Since the condition is monotonic in K (if it holds for K, it holds for any smaller K), we can use binary search. To process all queries efficiently, use **parallel binary search**.

Solution Approaches

Method 1: Euler Tour + Segment Tree

- For each color k, maintain a segment tree storing the sum of hash values of vertices of that color.
- Process vertices in DFS order, adding/subtracting hash values appropriately when entering/leaving a vertex.
- Need to handle LCA carefully when adding vertex values.

- Complexity: $O((N+Q)\log^2 N)$.
- A similar approach can be implemented using rolling hash.

Method 2: HLD + Segment Tree

- Maintain a segment tree over the heavy-light decomposition, where leaf k stores the hash value of vertex k (or 0 if not yet active).
- Activate vertices in increasing order of color, updating the segment tree.
- For each query, compare the sum of values on the two paths.
- Complexity: $O((N+Q)\log^3 N)$, but with BIT and careful implementation, it can pass.

Summary

- The solution idea is relatively straightforward, but implementation is heavy.
- It's useful to have well-tested library code for tree path queries (HLD/Euler tour + segment tree).
- Implementation choices affect code length and constant factors significantly.
- Team assignment for implementation may greatly affect solving time.
- Note: An $O((N+Q)\log N)$ solution also exists.

Problem I. Infinite Golf

The numbering of cells in the grid follows a classic diagonal zig-zag pattern. We can analyze the structure by grouping the cells based on their diagonal.

Each diagonal consists of cells where the sum of the coordinates is constant. Let $S = r + c$ be the sum of the row and column indices of a cell (r, c).

- The first diagonal ($S = 2$) contains only cell $(1, 1)$.
- The second diagonal ($S = 3$) contains cells $(1, 2)$ and $(2, 1)$.
- The third diagonal ($S = 4$) contains cells $(3, 1)$, $(2, 2)$, and $(1, 3)$.
- In general, the d-th diagonal (where $d = S - 1$) contains exactly d cells.

1 Mathematical Formulation

To find the value of a cell at (r, c), we can split the calculation into two parts:

1. Find the number of cells in all diagonals strictly preceding the current one.
2. Determine the relative position of the cell within the current diagonal.

1.1 Step 1: Sum of Preceding Diagonals

For a cell with coordinate sum $S = r + c$, all preceding diagonals have coordinate sums from 2 to $S - 1$. The number of cells in these diagonals is the sum of the first $S - 2$ integers:

$$N_{\text{prev}} = \sum_{i=1}^{S-2} i = \frac{(S-2)(S-1)}{2}$$

1.2 Step 2: Position Within the Current Diagonal

Depending on whether S is even or odd, the traversal direction along the diagonal changes:

- **Case 1: S is odd**
  The path moves downwards and to the left (from top-right to bottom-left). The values increase as the row index r increases.
  $$\text{Value} = N_{\text{prev}} + r$$
  For example, for $(1, 2)$, we have $S = 3$ (odd) and $N_{\text{prev}} = 1$. The value is $1 + 1 = 2$.
- **Case 2: S is even**
  The path moves upwards and to the right (from bottom-left to top-right). The values increase as the column index c increases.
  $$\text{Value} = N_{\text{prev}} + c$$
  For example, for $(3, 1)$, we have $S = 4$ (even) and $N_{\text{prev}} = 3$. The value is $3 + 1 = 4$.

2 Implementation Details

Since $r, c \leq 10^9$, their sum S can be up to $2 \cdot 10^9$. Computing $N_{\text{prev}} \approx \frac{S^2}{2}$ requires a data type that can hold values up to $2 \cdot 10^{18}$. A standard 64-bit signed integer (`long long` in C++) is sufficient.

To prevent overflow when calculating $(S-2)(S-1)$ prior to division, we should check which of the two consecutive terms is even and divide it by 2 first:

$$\text{If } (S-2) \text{ is even:} \quad N_{\text{prev}} = \left(\frac{S-2}{2}\right) \cdot (S-1)$$

$$\text{If } (S-2) \text{ is odd:} \quad N_{\text{prev}} = (S-2) \cdot \left(\frac{S-1}{2}\right)$$

This guarantees $O(1)$ time complexity and $O(1)$ auxiliary space complexity.

Problem J. Jigsaw With Words

The problem asks us to find all occurrences of W given words in an $N \times N$ grid of characters, where adjacent characters can share an edge or a vertex (8-directional movement). A single path representing a word cannot reuse the same cell. Once all occurrences of all words are found, the cells belonging to these occurrences are deleted, and the remaining letters are concatenated in row-major order to produce the final string.

The constraints are:

- Grid size: $1 \leq N \leq 10$, meaning the total number of cells is $N^2 \leq 100$.
- Word length: $L \leq 10$.
- Number of queries: W.
- Number of test cases: $T \leq 12$.

3 Why Naive DFS Fails

A naive Depth First Search (DFS) would attempt to find each word S by starting from every cell (r, c) and exploring all 8 neighboring directions. In the worst-case scenario (for example, a grid filled entirely with the letter '`a`' and a query word consisting of 10 `a`'s), the number of paths of length 10 starting from a single cell can be up to $8^9 \approx 1.34 \cdot 10^8$. Across 100 cells and multiple words, this approach will easily exceed the 1-second time limit.

4 Meet-in-the-Middle Approach

To optimize the search, we can split the word S of length L into two halves:

1. **Prefix**: The first $L_1 + 1$ characters of S, where $L_1 = \lfloor L/2 \rfloor$.
2. **Suffix**: The remaining $L_2 = L - L_1$ characters of S.

These two paths will overlap at exactly one cell, which we refer to as the *junction cell* (corresponding to character $S[L_1]$).

4.1 Step 1: Path Generation

- We run a DFS from every cell to find all valid paths of length $L_1 + 1$ matching the prefix $S[0 \dots L_1]$. We store these paths in a list `prefixes[C]`, where C is the ending (junction) cell of the prefix path.
- Similarly, we run a DFS from every cell to find all valid paths of length L_2 matching the suffix $S[L_1 \dots L-1]$. We store these paths in a list `suffixes[C]`, where C is the starting (junction) cell of the suffix path.

4.2 Step 2: State Representation using Bitmasks

To ensure that a combined prefix and suffix path form a valid simple path, we must verify that they do not share any cells other than the junction cell.

Since $N^2 \leq 100$, we can represent the set of visited cells in any path using a bitmask of size 100. We can implement this using two 64-bit integers:

- `mask_lo`: A bitmask for cells with IDs from 0 to 63.
- `mask_hi`: A bitmask for cells with IDs from 64 to 99.

4.3 Step 3: Merging Paths

For each cell C on the board, we pair every prefix path in `prefixes[C]` with every suffix path in `suffixes[C]`. Two paths can be merged if and only if their masks overlap exclusively at the cell C:

$$(\text{prefix.mask\_lo} \;\&\; \text{suffix.mask\_lo}) = \text{shared\_lo}$$

$$(\text{prefix.mask\_hi} \;\&\; \text{suffix.mask\_hi}) = \text{shared\_hi}$$

where `shared_lo` (or `shared_hi`) has only the bit corresponding to cell C set to 1.

If a valid merge is found, all cells in both the prefix and suffix paths are marked in a global boolean array `global_marked`.

5 Complexity Analysis

- **Time Complexity**: By splitting the search, the DFS depth is halved. The maximum number of states explored per start cell is reduced from $O(8^L)$ to $O(8^{L/2})$. For $L = 10$, this reduces the worst-case operations per start cell from $\approx 10^8$ to $\approx 3.2 \cdot 10^4$. The merging phase takes $O(|prefixes[C]| \cdot |suffixes[C]|)$ time, which is extremely fast in practice.
- **Space Complexity**: We store paths of length at most $\lceil L/2 \rceil + 1 \leq 6$ cells. For each path, we store its bitmasks and the list of cells. The memory consumed per test case is negligible and well within the 1024 MB limit.

Problem K. King's Skyscraper

We are given M elevators. For each elevator, we can make moves of size $+g$ (up) and $-d$ (down). We must make exactly N moves in total, starting from floor 0. We cannot visit any negative floor during the process. We want to find the minimum positive floor ($F > 0$) reachable at the end of the N-th move using any single elevator.

Let u be the number of up moves ($+g$). The number of down moves ($-d$) must then be $N - u$, where $0 \le u \le N$. The final floor F after N moves is given by:

$$F = u \cdot g - (N - u) \cdot d = u(g + d) - N \cdot d$$

6 Mathematical Derivation

We want to find the smallest possible final floor F such that $F > 0$.

Using the equation for F:

$$u(g + d) - N \cdot d > 0$$

$$u(g + d) > N \cdot d$$

$$u > \frac{N \cdot d}{g + d}$$

Since u must be an integer, the smallest valid integer value for u is:

$$u_{\min} = \left\lfloor \frac{N \cdot d}{g + d} \right\rfloor + 1$$

Note that since $g \ge 1$ and $d \ge 1$, we have:

$$u_{\min} \le \frac{N \cdot d}{g + d} + 1 < N + 1 \implies u_{\min} \le N$$

This guarantees that $u_{\min}$ is always a valid number of up moves (i.e., we do not exceed the total budget of N moves).

7 Feasibility of Intermediate States

The problem states that we cannot go below floor 0 at any intermediate step. We must verify if there exists a permutation of the $u_{\min}$ up moves and $N - u_{\min}$ down moves such that all prefix sums are non-negative.

By Raney's Lemma (a generalization of the Cycle Lemma), for any sequence of steps whose total sum is positive ($F > 0$), there exists at least one cyclic shift of this sequence such that all prefix sums starting from the shift origin are strictly positive (except for the initial state, which is 0).

Since we can choose the order of our N button presses arbitrarily, the intermediate non-negativity constraint is automatically satisfied as long as the final floor $F > 0$. Hence, we only need to compute $u_{\min}$ and find the resulting floor F.

8 Algorithm and Complexity

For each of the M elevators, we can compute the minimum positive floor in $O(1)$ time:

1. Compute $u_{\min} = \lfloor \frac{N \cdot d}{g + d} \rfloor + 1$.
2. Compute the final floor $F = u_{\min}(g + d) - N \cdot d$.

3. Find the minimum F among all M elevators.

The overall time complexity is $O(M)$, and the space complexity is $O(1)$. Since $N \leq 10^6$ and $M \leq 2000$, this approach is extremely fast and will execute in less than a millisecond.

Problem L. Let's Find The Sum

Problem Overview

Given an integer N, compute

$$S(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \left\lfloor \frac{N}{ij} \right\rfloor.$$

Solution 1

Let:

- $d(n)$: number of positive divisors of n.
- $D(n) = \sum_{k=1}^{n} d(k)$.

Then,

$$S(N) = \sum_{k=1}^{N} d(k) \left\lfloor \frac{N}{k} \right\rfloor = \sum_{k=1}^{N} d(k) \sum_{i=1}^{\lfloor N/k \rfloor} 1 = \sum_{ik \leq N} d(k) = \sum_{k=1}^{N} D\left(\left\lfloor \frac{N}{k} \right\rfloor \right).$$

Split the sum by the size of k relative to $N^{1/3}$:

- For small k, compute $\lfloor N/k \rfloor$ directly (each in $O(\sqrt{N/k})$).
- For large k, precompute $D(\cdot)$ and group by the value of $\lfloor N/k \rfloor$.

Total complexity: $O(N^{2/3})$.

Solution 2

Observe that the desired sum equals the number of ordered triples (a, b, c) of positive integers such that $abc \leq N$. This is a known problem (e.g., AtCoder ABC227 C). Under the constraint $a \leq b \leq c$, it can be counted in $O(\sqrt[3]{N})$ or $O(N^{2/3})$ as well.