Problem Analysis

Disclaimer: *This is an analysis of some possible ways to solve the problems of The 2024 ICPC Asia Jakarta Regional Contest. Since the purpose of this analysis is mainly to give the general idea to solve each problem, we left several (implementation) details in the discussion for reader's exercise. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help.*

	Problem Title	Problem Author
A	Scrambled Scrabble	Ashar Fuadi
B	ICPC Square	Pikatan Arya Bramajati
C	Saraga	Pikatan Arya Bramajati
D	Aquatic Dragon	Ashar Fuadi
E	Narrower Passageway	Ashar Fuadi
F	Grid Game 3-angle	Prabowo Djonatan
G	X Aura	Lie, Maximilianus Maria Kolbe
H	Missing Separators	Ammar Fathin Sabili
I	Microwavable Subsequence	Pikatan Arya Bramajati
J	Xorderable Array	Prabowo Djonatan
K	GCDDCG	Muhammad Ayaz Dzulfikar
L	Buggy DFS	Muhammad Ayaz Dzulfikar
M	Mirror Maze	Rafael Herman Yosef

Analysis Authors

Ammar Fathin Sabili

Lie, Maximilianus Maria Kolbe

Pikatan Arya Bramajati

Prabowo Djonatan

Rafael Herman Yosef

A. Scrambled Scramble

Let's ignore **NG** and **Y** case for now. Assuming that we have v vowels and c consonants, and s is the number of syllables. The number of syllables we can make is $s = \min(v, \lfloor \frac{c}{2} \rfloor)$, and the word length is $3 \times s$.

To solve this task, we count the number of vowels (excluding **Y**), consonants (excluding **Y**, **N**, **G**), **Y**, **N**, and **G**. Then, we iterate over the number of **NG** consonants we want to use. For each **NG**, we reduce the number of **N** and **G** by one. For each possible number of **NG** consonants, we try different combinations of **Y** as vowels and consonants.

For each scenario, we calculate the word length that we can construct. When calculating the length, we adjust the formula to account for **NG** consonants, prioritizing them over other consonants. Thus the length of the word for each scenario is $3 \times s + \min(2 \times s, |NG|)$

The time complexity of this solution is $O(N^2)$.

There's also a solution that runs in linear time.

B. ICPC Square

First, notice that we can never visit any floor that isn't a multiple of S. This means that we can divide both N and D by S and we start at floor 1. We will multiply our answer with S in the end to get our final answer. Let N' = $\lfloor \frac{N}{S} \rfloor$ and D' = $\lfloor \frac{D}{S} \rfloor$.

The upper bound floor that can be reached is $\min(2 \times D', N')$. If we try to reach any floors higher than that, then the difference between that floor and the earlier floor will be bigger than D'. This is because the factor of a number that is not itself must be at most half of itself.

Let Y = $\min(2 \times D', N')$. We can reach floor Y if any of these conditions satisifed.

- Y is even. We can reach floor Y from $\frac{Y}{2}$, and because $\frac{Y}{2}$ value is at most D', therefore we can reach floor $\frac{Y}{2}$ from 1.
- There exists an integer X such that Y is divisible by X, and $Y - \frac{Y}{X} \leq D'$. $\frac{Y}{X}$ value is also at most D', thus we can reach floor $\frac{Y}{X}$ from floor 1. We can find X in $O(\sqrt{Y})$.

If we cannot reach floor Y, then the answer is floor $Y - 1$ because that floor is even.

The time complexity of this solution is $O(\sqrt{N})$

C. Saraga

For strings, denote the $+$ operation as concatenation.

Notice that for a string Z to be an interesting abbreviation of S and T, it must consists of three non-empty string parts P, Q, and R such that:

- $Z = P + Q + R$
- $P + Q$ is a prefix of S.
- $Q + R$ is a suffix of T.

Consider a valid triple (P, Q, R) where $|Q| > 1$. We can always construct two new strings:

- Q' is taking Q and removing its last character.
- R' is taking R and appending that last character of Q to the beginning of R.

Since $P + Q$ is a prefix of S, that means $P + Q'$ is also a prefix of S. Next, $Q' + R' = Q + R$, so it's also a suffix of T. Both Q' and R' are non-empty. Therefore, (P, Q', R') is also a valid triple.

From any valid triple, we can always repeatedly do the same process to get a valid triple with the same length where $|Q| = 1$. That means, to consider for all possible triples, we can just consider all triples with $|Q| = 1$.

If $|Q| = 1$, then Q can only be a single letter, which there are only 26 possibilities. Since P, Q, and R can't be empty, we can only consider prefixes of S and suffixes of T with a length of at least 2. For each such prefix of S, look at its last character. For each such suffix of T, look at its first character. These two characters will be the single-letter string Q, so they must match.

For each letter from `a` to `z`, find its shortest prefix in S and its shortest suffix in T to form Z. Out of all 26 letters, find any one with shortest length, or report if Z can't be formed from any of the 26 letters.

The time complexity of this solution is $O(N)$.

D. Aquatic Dragon

If $T_f \leq T_s$, then we can just fly all the way from island 1 to island N without swimming or walking. From now on, we will solve if $T_f > T_s$.

For a valid solution, if we look at the walking steps, it will consist of a set of segments. Since each tunnel can only be traversed once, there can't be any two walking segments that overlap. However, two walking segments can still touch at their endpoints.

There can be islands that are not covered by any walking segments. We can add a zero-length walking segment for each of these islands, which will help in generalizing the steps in the entire journey.

After doing that, we can decompose the entire journey into several steps, where each step going from some island x to some island y ($x < y$) corresponds to exactly one walking segment, which is as follows:

1. Walk from island x to either island $y-1$ or island y, then go back to island x to activate every shrine in that area.
2. Move the dragon to island y.

We walk to island $y-1$ if the walking segment doesn't touch the next walking segment. We walk to island y if the walking segment touches the next walking segment.

Note that after walking from x to $y-1$ and back, we activate every single shrines from x to $y-1$. That means, our dragon won't get any stamina increase when going from x to y. That means, when going from x to $y-1$, we have to swim all the way. When going from $y-1$ to y, we can choose to either fly or swim.

In a step where the walking segment touches the next walking segment, it's always not optimal if we swim at the end of that step. That's because, if we do that, then it's just better to merge that walking segment with the next walking segment to activate more shrines earlier without any consequences.

That means, each step for going from island x to island y is of three types:

1. Walk to $y-1$ and back, swim at the end.
2. Walk to $y-1$ and back, fly at the end.
3. Walk to y and back, fly at the end.

For now, let's assume we can't fly at all. If we want to begin a step at island x, before we activate the shrine at island x, the stamina is always $P_1 + P_2 + ... + P_{x-1} - D \times (x-1)$ no matter what we did before this. That means, if we want to do a step from island x to island y where we swim in the end, it must hold that $P_1 + P_2 + ... + P_{y-1} - D \times (y-1) \geq 0$. The total time of this step is $T_w \times (y-1-x) \times 2 + T_s \times (y-x)$.

Let's add flying into the mix. The only thing that matters is the last time we flew and whether that step was type 2 or type 3. Let's say the last time we flew was flying from island $l-1$ to island l and it was a type 2 step. If we want to begin a step at island x, before we activate the shrine at island x, the stamina is always $P_l + P_{l+1} + ... + P_{x-1} - D \times (x-l)$ because flying makes our stamina go down to 0. The requirement to do the step as before is similar, it's just $P_l + P_{l+1} + ... + P_{y-1} - D \times (y-l) \geq 0$.

If the last time we flew was a type 3 step, then the shrine at island l is already used after that step, so the stamina at island x is $P_{l+1} + P_{l+2} + ... + P_{x-1} - D \times (x-l)$. The requirement becomes $P_{l+1} + P_{l+2} + ... + P_{y-1} - D \times (y-l) \geq 0$.

We can make a prefix sum of P. Let $Q_i = P_1 + P_2 + ... + P_i$. To simplify the many calculations after this, define two more arrays R and S such that $R_i = Q_{i-1} - D \times (i-1)$ and $S_i = Q_i - D \times (i-1)$. If the last time we flew was flying from island $l-1$ to island l with a type 2 step and we want to begin a step at island x to island y where we swim in the end, it must hold that $R_y \geq R_l$. If it was a type 3 step, then it must hold that $R_y \geq S_l$.

We can calculate a value pivot which can be R_l (if the last time we flew was a type 2 step) or S_l (if the last time we flew was a type 3 step). Then, doing a type 1 step from x to y requires $R_y \geq$ pivot.

Next, consider the type of step where we fly in the end. Let's consider a type 2 step first. The stamina at island $y-1$ must be strictly positive. It can be obtained that the equivalent requirement is $R_y >$ pivot $- D$ must hold. For a type 3 step, it must hold that $S_y >$ pivot $- D$.

Notice that after doing that step, the last time we fly changes, and a new pivot value is calculated, which is actually either R_y or S_y depending on which type the step is. That means, doing the requirement to do the step above is newPivot $>$ pivot $- D$.

Let's solve the problem if flying is not allowed. To do a step from any island to an island y, the only requirement is that $R_y \geq R_1$. The swimming time is fixed, we only have to minimize the walking time. Let's say we add $T_w \times (N-1) \times 2$ to the total walking time. Doing a step essentially reduces the walking time by $T_w \times 2$. That means, we need to maximize the number of steps. We can do a step to any island y that has $R_y \geq R_1$. That means, the maximum number of steps is the total number of y ($1 < y \leq N$) with $R_y \geq R_1$. Note that there is a corner case when it comes to island N. We need to consider the case where we go to island N and back before moving our dragon to island N by swimming all the way. We can do this if $S_N \geq R_1$.

After knowing everything, we can solve this problem using dynamic programming from the back. Let dp$[x][0]$ be the minimum time to go to island N if we're in island x with our dragon and the last time we flew was from island $x-1$ to island x with a type 2 step. Let dp$[x][1]$ be the same thing, but for type 3. For both, we can calculate the value of pivot. When trying to calculate the optimal value for dp$[x][e]$, let's say we want to consider a solution where the next time we fly is from island $y-1$ to island y. That means, our journey from island x to island y must end with a step where we fly at the end. Depending on the type of that step, we can calculate the value of newPivot and find out whether the next DP value dp$[y][e']$ is $e' = 0$ or $e' = 1$. That means, it must hold that newPivot $>$ pivot $- D$. If that holds, the maximum number of steps where we swim at the end is the number of islands i ($x < i < y$) with $R_i \geq$ pivot. Let's say that number is c. That means, the minimum time for this case is $T_s \times (y - 1 - x) + T_f + T_w \times (y - 1 - x - c + e') \times 2 +$ dp$[y][e']$. We try for both cases for the two possible types for the last step going into island y. The naive implementation of this solution runs in $O(N^2)$.

Let's optimize it. Notice that the hard thing to keep track is the value c, because it's heavily reliant on the value of pivot and there are a total of $2N$ different values of pivot. But we can handle it with one important observation. To get dp$[x][e]$ from dp$[y][e']$, the only requirement is that newPivot $>$ pivot $- D$ must hold. However, is it optimal to get dp$[x][e]$ from dp$[y][e']$ if newPivot $\geq$ pivot? Consider the last step at that case, which is when we fly at the end. Consider what happens if we change that step such that we swim at the end. Since we assume that $T_f > T_s$, changing that step will reduce the time. Next, look at what happens after the step. Flying at the end changes our main comparison value to newPivot, while swimming keeps it being pivot. If newPivot $\geq$ pivot, it's essentially worse because it reduces our move possibilities and reduces the number of steps we can do to reduce the total walk time. That means, flying to an island y where newPivot $\geq$ pivot is objectively worse and we can be good just by ignoring that case.

That means, for each dp$[x][e]$, we can get that value from every dp$[y][e']$ such that pivot $- D <$ newPivot $<$ pivot. Since the way we calculate pivot is the same as newPivot, that means, we can sort the pairs (x, e) based on increasing values of pivot (either R_x or S_x). We iterate each pair (x, e) based on that order and calculate its value of dp$[x][e]$.

In order to get the values of dp$[x][e]$, we maintain two segment trees, for $e' = 0$ and $e' = 1$. Each index y in the segment tree maintains the value of dp$[y][e']$ plus the additional values of T_s, T_f, and T_w if we want to take its value for dp$[x][e]$, so we can do a range minimum query to get the optimal value for the transition.

The following is the way to handle each value of T_s, T_f, and T_w:

- For T_s, we just add each index y in the segment tree with $T_s \times (y - 1)$, so we can add $T_s \times x$ after the range query.
- For T_f, it's just constant since we always do it once in each transition, so add everything by T_f.
- For T_w, initially, we don't walk at all since $R_y \geq$ pivot always holds if pivot is from the smallest value. Each time we iterate (x, e) to a bigger value of pivot, there can be more values of R_x that becomes smaller than the current value of pivot, so for each of them, we must do 1 length worth of walking in order to pass that, so for every index y in the segment tree after that x, we add $T_w \times 2$. We handle that using a range update. Additionally, there's also an extra $T_w \times 2$ for each value with $e' = 1$.

After we get a new value of dp$[x][e]$, we add it to the corresponding segment tree. Each time we iterate to the next bigger value of pivot, we have to discard the values of previously calculated dp$[y][e']$ with newPivot $\leq$ pivot $- D$ by changing its value in the segment tree to infinity.

The answer is dp$[1][0]$. Don't forget to handle the corner case with island N as previously mentioned.

The time complexity of this solution is $O(N \log N)$.

E. Narrower Passageway

We can solve this problem with contribution technique. Let prob(r, c) be the probability of cell (r, c) contributing to the expected total strength, i.e. there is a connected area such that the strength is taken from cell (r, c). The expected total strength will be $\sum P_{r,c} \cdot \text{prob}(r, c)$.

For a cell (r, c) to contribute to the expected total strength, the connected area $[u, v]$ has to satisfy the following conditions:

- $P_{r,c}$ is the largest power in row r of the connected area, and
- the maximum power in the other row of the connected area is larger than $P_{r,c}$.

The main challenge of this problem is to compute the number of such connected areas for all cells efficiently. We can use sweep-like algorithm to compute these values in the order of $P_{r,c}$ sorted ascendingly. Let's call a cell that has been computed by the sweep as *active*. We need to maintain ranges consisting of only active cells, as well as the maximum value for each row in the connected areas. This can be achieved in $O(1)$ or $O(\log N)$ using Union-Find Disjoint Set or STL Set.

Suppose that we want to calculate the contribution of cell (r, c). Let r' be the row other than r. Denote $[\text{left}_1, \text{right}_1]$ as the maximal range consisting of active cells such that $\text{left}_1 \le c \le \text{right}_1$. Denote $[\text{left}_2, \text{right}_2]$ as a range that satisfy the following conditions:

- left_2 is the maximum between left_1 and $p + 1$ with p being the rightmost column to the left of c such that $P_{r',\text{left}_2} > P_{r,c}$.
- right_2 is the minimum between right_1 and $p - 1$ with p being the leftmost column to the right of c such that $P_{r',\text{right}_2} > P_{r,c}$.

The value of left_2 and right_2 can be calculated in $O(\log N)$ using binary search or STL Set lower bound.

Note that $P_{r,c}$ contributes for all connected areas $[u, v]$ that satisfy one of the following conditions:

- $\text{left}_1 \le u < \text{left}_2$ and $c \le v \le \text{right}_1$, or
- $\text{left}_1 \le u \le c$ and $\text{right}_2 < v \le \text{right}_1$.

Do not forget to handle the double counting if both conditions are satisfied.

Finally, the cells outside the connected area $[u, v]$ will not affect the contribution. Denote $f(r, c, u, v)$ as the probability of the connected area $[u, v]$ such that cell (r, c) contributes to the expected total strength. Column $u-1$ and $v+1$ has to be foggy. Columns in $[1, u-2]$ and $[v+2, N]$ can be either foggy or not. Therefore, $f(r, c, u, v)$ is 2 to the power of the number of columns in $[1, u-2]$ and $[v+2, N]$, divided by 2^N. Do not forget to handle the case if $u = 1$ or $v = N$.

Therefore, the contribution of $P_{r,c}$ is the sum of $f(u, v)$ for all valid connected areas $[u, v]$. This value can be precomputed and we can count prob(r, c) in $O(1)$. This can be done by using calculating the sum for each of the left and right sides independently, and then multiplying the two sums. To calculate each side, we can calculate the sum of powers of 2 in $O(1)$ using the identity $2^0 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1$.

Do not forget to handle the double counting due to the value of $P_{r,c}$ being not distinct. This can be handled easily in the sweep, for example, by working on the smaller c first, then add the following condition: $P_{r,c}$ is strictly larger than $P_{r,c'}$ for active cells (r, c') that satisfy $c' > c$.

The time complexity of this solution is $O(N \log N)$.

F. Grid Game 3-angle

For each $0 \le i \le K$, let a_i be the result of considering every cell (x, y) satisfying $x \equiv i \pmod{(K+1)}$, taking the number of stones modulo $K+1$ in each of them, and XOR-ing all of those modulo results.

Theorem 1. If any a_i $(0 \le i \le K)$ is not zero, then the first player will win the game. Otherwise, the second player will win the game.

This theorem essentially solves the problem, so we will try to prove it by introducing more lemmas.

Lemma 2. If all a_i $(0 \le i \le K)$ is zero, then whatever move made by the next player will make at least one j $(0 \le j \le K)$ such that a_j becomes non-zero.

Proof. Suppose the player choose cell (x, y), then the number of stones (modulo $K+1$) on that cell must also change (recall that the player is only allowed to remove 1 to K stones). Consequently, the value of $a_{x \bmod (K+1)}$ must also change. The cell (x, y) is the only cell that affects $a_{x \bmod (K+1)}$, because for each of all other cells, say (x', y'), in which the player can add stones to it, none of them is $x' \equiv x \pmod{(K+1)}$ because $x < x' \le x + K$. □

Lemma 3. If there exists an i with $a_i \neq 0$, then there exists a move that makes every a_i becomes zero.

Before we start proving this lemma, we need to see how you can control the value of a_i when you remove and add stones.

Lemma 4. Suppose a_i is not zero (for some $0 \leq i \leq K$), then among every cell (x, y) with $x \equiv i \pmod{(K+1)}$, we can choose exactly one cell such that after removing (from 1 to K stones) from it, the value of a_i becomes zero.

Proof. This is the consequence of the result of the standard nim game where all of the piles have less than or equal to K stones. □

Lemma 5. Suppose you have a sequence $x_1, \ldots, x_n$ such that $n \geq 2$ and $0 \leq x_i \leq K$, then it is possible to modify any two elements (into a value that is also at most K) which result in their XOR sum to be zero.

Proof. Pick any two elements, p and q, then let $X = x_1 \oplus \ldots \oplus x_n \oplus p \oplus q$. Let b be the position of the most significant bit of K. If the b-th bit is set in X, then we set $p' = 2^b$, otherwise $p' = 0$. Next, set $q' = X \oplus p'$. Do notice that the b-th bit of q' is guaranteed to be 0, so $q' \leq K$. Finally, replace p with p' and q with q' and we are done. □

Note that modifying one element is indeed not enough, as K is not a power of two minus one. Now we are ready to prove our remaining lemmas.

Proof of Lemma 3. Let I be the set of all i such that $a_i \neq 0$. Let c_i ($i \in I$) be a cell whose stone removals can make a_i into 0 (such a cell must exist by Lemma 4). Among all the cells c_i, pick the one that has the lowest row number (such a cell uniquely exists), and denote it as C. Start the move by removing stones from C, making its corresponding a_i become zero. Next, for all the remaining $a_i \neq 0$, there are at least two corresponding cells which we can add stones to them (this is due to the setting of the game, and the fact that we chose the cell with the lowest row number hence these two cells exist "below" the chosen cell). Since we can add 0 to K stones, we can essentially "modify" the number of stones (modulo $K+1$) of that cell to any value at most K; hence by Lemma 5, we can make the corresponding a_i zero. □

To complete the main theorem.

Proof of Theorem 1. If the current grid has no more stones, every a_i will be zero, hence it is a losing position. Otherwise, if all a_i is zero, then by Lemma 2, the next turn will have at least one a_i that is non-zero. If there exists an a_i that is positive, then by Lemma 3, there exists a move such that the next turn will be the losing position. Since the game eventually ends, if the first player starts with at least one a_i that is positive, then there is a sure-win strategy. □

This completes the solution.

The time complexity of this solution is $O(M \log M)$ for each test case.

G. X Aura

Note that for all scenarios to be valid, the total penalty of all cycles have to equal 0. Define this property as *zero-cycle*.

The most important observation is that the grid is *zero-cycle* iff all two-by-two subgrids within the grid are *zero-cycle*. It can be proven that any cycle can be transformed to any other cycle while maintaining the total penalty of 0 by a series of substitutions to the penalty. Thus, we can check whether a grid is *zero-cycle* in $O(NM)$.

Let $d(u, v)$ be the minimum total penalty from cell (r_u, c_u) to (r_v, c_v). Note that due to the *zero-cycle* property, the following equations hold: $d(u, v) = -d(v, u)$ and $d(u, k) + d(k, v) = d(u, v)$. To summarize, the calculation of minimum total penalty behaves like vector addition.

There are a lot of solution, and the following is one of the easier to implement. Denote s as the cell $(1, 1)$. Note that $d(u, v) = d(u, s) + d(s, v) = d(s, v) - d(s, u)$, which makes this problem a single-source shortest path problem. We can precompute $d(s, u)$ in $O(NM)$ and answer each query in $O(1)$.

The time complexity of this solution is $O(NM)$.

H. Missing Separators

Greedy won't work for this problem, let's use dynamic programming instead.

First, assume that we want to construct the words from the front to the end of S. Let DP$[x][y]$ denote how many more words we can construct if the last word we constructed is the substring $S[x..y]$. Therefore, we will only consider the substring after the y-th position for the transitions. Note that it is very possible that the DP will return an impossible case, in which the list couldn't be in a lexicographical order later on.

For the transitions, for some z, we need to guarantee that substring $S[y + 1..z]$ comes after $S[x..y]$ in a lexicographical order. It is important to note that if $S[y + 1..z]$ comes after $S[x..y]$, then actually $S[y+1..z+1]$ also comes after $S[x..y]$. In fact, this is also applicable for all $k \geq 0$ that $S[y+1..z+k]$ comes after $S[x..y]$. Thus, given x and y, we can try to find the first possible z.

Observe that we can do it by finding the longest common prefix of $S[x..y]$ and $S[y + 1..]$. If the longest common prefix is $S[x..y]$ itself, then $S[y + 1..]$ will always come after $S[x..y]$, given that the new word is longer. Else, we only need to check their first different letter. We make sure that the new word has a lexicographically larger letter, or otherwise it will lead to an impossible case.

But how do we find the longest common prefix of $S[x..y]$ and $S[y + 1..]$ in an efficient way? We

might think of doing a binary search, possibly with string hashing, but the overall runtime will be at least $\Omega(|S|^2 \log |S|)$ which is still quite slow. Another idea is to precompute these longest prefixes for every possible substring of S. The main trick here is that we can compute the **longest common prefix of suffixes** of S using another dynamic programming!

Let LCPS$[p][q]$ denote the length of the longest common prefix of the suffixes $S[p..]$ and $S[q..]$. It's not hard to see that LCPS$[p][q]$ is simply $1 +$ LCPS$[p+1][q+1]$ if $S[p] = S[q]$, or 0 otherwise. Utilizing this second DP, the longest common prefix of $S[x..y]$ and $S[y+1..]$ will then have a length of LCPS$[x][y+1]$, subject to be minimized with the length of $S[x..y]$ itself.

Finally, after finding the z, we still need to find the maximum DP value among DP$[y+1][z+k]$ for all $k \geq 0$. This can be done in $O(1)$ by (again) precomputing the max suffix of the second state of the DP.

After getting the DP values, we can use these DP values to backtrack to get a possible valid construction for the answer.

The time complexity of this solution is $O(|S|^2)$.

I. Microwavable Subsequence

Define a value x as occuring if and only if it's present in A; otherwise, we define it as non-occuring. Let c be the number of occuring values.

For each pair x and y, if both x and y are non-occuring, then $f(x, y) = 0$. If exactly one of x or y is occuring, then $f(x, y) = 1$. The number of such pairs is $c \times (M - c)$.

Now consider the case where both x and y are occuring. For now, let's ignore all of the numbers in A aside from x and y. The length of the longest alternating subsequence is equal to the number of pairs of **adjacent elements with different values**, plus one.

Now, we want to calculate the number of pairs for all (x, y) at once. The number of such pairs is equal to the number of pair of indices (i, j) where $i < j$, and all of the elements within the range from $i + 1$ to $j - 1$ is neither A_i nor A_j.

To calculate that number, we can iterate A from left to right while maintaining the index of the last occurence for each value. For each index i, let l be the index of the last occurence of A_i. We want to calculate the number of last occurences between $l + 1$ and $i - 1$ inclusive. We can solve this using Fenwick Tree/segment tree data structure.

Don't forget to add one for each pair of occuring values x and y.

The time complexity of this solution is $O(N \log N)$.

J. Xorderable Array

Let's say we have two non-negative integers B_0 and B_1. Let's observe how to compare the values of B_0 and B_1 by their binary representations bit by bit. We iterate the bits of B_0 and B_1 simultaneously from the most significant bit to the least significant bit. Let's say the current bit of each of B_0 and B_1 are b_0 and b_1 respectively. There are three cases:

- If $b_0 < b_1$, then it's decided that $B_0 < B_1$.
- If $b_0 > b_1$, then it's decided that $B_0 > B_1$.
- If $b_0 = b_1$, then we check the next bit.

Now, given two non-negative integers P and Q, let's consider the (P, Q)-xorderability of just one pair of values B_0 and B_1. Just like before, we iterate the bits simultaneously from the most significant bit. Let's say the current bit of each of P, Q, B_0, and B_1 are p, q, b_0, and b_1.

- If $p = q = 0$, then the comparison for b_0 and b_1 uses the same rule as an ordinary comparison.
- If $p = q = 1$, then the comparison for b_0 and b_1 uses the same rule as an ordinary comparison, but the verdict for $b_0 < b_1$ and $b_0 > b_1$ are flipped.
- If $p \neq q$, then there are two cases:
  - If $b_0 = b_1$, then it's actually impossible to xorder the pair (B_0, B_1). This is because no matter how we xorder the pair, there will always be the expression $1 \leq 0$ in this bit.
  - If $b_0 \neq b_1$, then no matter how we xorder B_0 and B_1, we will always have the comparison bits of B_0 and B_1 (after XOR-ed by p and q) to be equal, so we always have to check the next bit.

For now, let's ignore how to xorder the pair (B_0, B_1) and only consider its xorderability possibility. When is the pair (B_0, B_1) xorderable? Notice that the only case where it's unxorderable is when $p \neq q$ and $b_0 = b_1$. Let's consider how we would check it when iterating the bits:

- If $p = q$ and $b_0 \neq b_1$, then it's decided that it's xorderable.
- If $p \neq q$ and $b_0 = b_1$, then it's decided that it's unxorderable.
- If either both $p = q$ and $b_0 = b_1$, or both $p \neq q$ and $b_0 \neq b_1$, then we check the next bit.

Observe the logic above. Notice the similarity between that and the logic of comparing two binary integers we discussed before. The process above is the same as comparing the two integers $P \oplus Q$ and $B_0 \oplus B_1$. The pair (B_0, B_1) is xorderable if and only if $P \oplus Q \leq B_0 \oplus B_1$.

If (B_0, B_1) is xorderable, how should we xorder the pair to satisfy the xorder condition? The only decider of the xorder of the pair is the case when $p = q$. The order of B_0 and B_1 should be flipped if and only if the deciding bit is at $p = q = 1$.

Let's extend this knowledge to the entire array A. That means, a requirement for A to be (P, Q)-xorderable is that for every pair (i, j) $(i < j)$, it must hold that $P \oplus Q \leq A_i \oplus A_j$. In other words, let minXOR be the minimum value of $A_i \oplus A_j$ among all pairs in A. Then the requirement is $P \oplus Q \leq$ minXOR.

However, if A satisfies that requirement, does that mean we can xorder the entire array A to satisfy the xorder condition? Consider the aforementioned conditions for the required xorder. We want it such that for every bit position with $p = q = 1$, we flip the order of that bit. We can obtain that it's equivalent to sorting A based on the values of $A_i \oplus (P \& Q)$. Sorting it like that will make every pair in A satisfy the xorder condition. That means, if A satisfies the xorderability condition $P \oplus Q \leq$ minXOR, it's always possible to xorder it, making the xorderability condition the only requirement.

To solve the original problem, we first calculate minXOR. It can obtained that the value of minXOR just by sorting A initially, calculating the XOR of adjacent values $A_i \oplus A_{i+1}$, and finding the minimum XOR among those, so it's can be done in $O(N \log N)$.

Now, we want to calculate the number of pairs (i, j) $(i < j)$ such that $X_i \oplus X_j \leq$ minXOR. We can solve this using a trie data structure. We iterate X_i from index 1 to M. After each iteration, we insert the value X_i into the trie. Before we insert, we query the trie to calculate the number of values w in the trie such that $w \oplus X_i \leq$ minXOR.

The time complexity of this solution is $O(N \log N + M \log X)$.

K. GCDDCG

Before anything, we precompute an array mul such that mul$[x]$ is the number of elements in A that's divisible by x. We can calculate this using sieve-like iterations over the frequency array of the values in A.

We also precompute the mobius function values for all values from 1 to N. Recall that the mobius function $\mu(x)$ is defined as follows:

- If x is divisible by a square number bigger than 1, then $\mu(x) = 0$.
- Else, then $\mu(x) = (-1)^c$ with c being the number of distinct prime factors of x.

First, let's count the number of non-empty subsets of A with a GCD of 1. The inclusion-exclusion logic of this calculation can be calculated using the mobius function. The number of subsets such

that its GCD is a multiple of x is $2^{\text{mul}[x]} - 1$. Using the logic of the mobius function, we can get that the answer for this is $\sum_{x=1}^{N}(2^{\text{mul}[x]} - 1) \times \mu(x)$.

Now, let's only count the number of ways such that the GCD of both decks are both 1. First, let's assume that the subsets of both decks are allowed to overlap. The number of ways for both subsets if the GCD of the two subsets must be a multiple of x and y respectively is $(2^{\text{mul}[x]} - 1) \times (2^{\text{mul}[y]} - 1)$. Then the answer is $\sum_{x=1}^{N}\sum_{y=1}^{N}(2^{\text{mul}[x]} - 1) \times (2^{\text{mul}[y]} - 1) \times \mu(x) \times \mu(y)$.

However, the subsets are not allowed to overlap. Now, let's calculate the number of ways for both subsets if the GCD of the two subsets must be a multiple of x and y respectively and they can't overlap. The only values where they might overlap are the values that are multiples of $\text{LCM}(x, y)$. For each of those values, it can either be one of the two subsets, or neither. Let $L = \text{LCM}(x, y)$. Then, the number of ways for the two subsets if the subsets can be empty is $3^{\text{mul}[L]} \times 2^{\text{mul}[x]+\text{mul}[y]-2\times\text{mul}[L]}$. We can just subtract the number of ways where one of the subsets is empty, which is $2^{\text{mul}[x]} + 2^{\text{mul}[y]} - 1$. Therefore, the number of ways such that the GCD of both decks are both 1 is $\sum_{x=1}^{N}\sum_{y=1}^{N}(3^{\text{mul}[L]} \times 2^{\text{mul}[x]+\text{mul}[y]-2\times\text{mul}[L]} - 2^{\text{mul}[x]} - 2^{\text{mul}[y]} + 1) \times \mu(x) \times \mu(y)$.

Let's find a way to calculate this quickly. To do this, we first calculate the value of $\sum_{x=1}^{N}\sum_{y=1}^{N}(2^{\text{mul}[x]} - 1) \times (2^{\text{mul}[y]} - 1) \times \mu(x) \times \mu(y)$. This can be calculated by calculating $\left(\sum_{x=1}^{N}(2^{\text{mul}[x]} - 1) \times \mu(x)\right)^2$. After that, consider the difference between the value of $3^{\text{mul}[L]} \times 2^{\text{mul}[x]+\text{mul}[y]-2\times\text{mul}[L]} - 2^{\text{mul}[x]} - 2^{\text{mul}[y]} + 1$ and the previous simpler value $(2^{\text{mul}[x]} - 1) \times (2^{\text{mul}[y]} - 1)$. Notice that we can rewrite $(2^{\text{mul}[x]} - 1) \times (2^{\text{mul}[y]} - 1)$ as $2^{\text{mul}[x]+\text{mul}[y]} - 2^{\text{mul}[x]} - 2^{\text{mul}[y]} + 1$. That means, the only difference is on the first term.

Let's calculate the sum of this difference. Notice that other than being dependent on x and y, this difference is also dependent on the value of L. Notice that if $L > N$, then $\text{mul}[L] = 0$ always holds. Then, $3^{\text{mul}[L]} \times 2^{\text{mul}[x]+\text{mul}[y]-2\times\text{mul}[L]}$ is equal to $2^{\text{mul}[x]+\text{mul}[y]}$, so there's no difference we need to calculate. So the only differences we need to calculate are for $L \leq N$.

We want to iterate every single pair (x, y) such that $L = \text{LCM}(x, y) \leq N$. We can break down L into a triple of values (G, x', y') such that:

- $G = \text{GCD}(x, y)$
- $x' = \frac{x'}{G}$
- $y' = \frac{y'}{G}$

Then we get that $L = G \times x' \times y'$. We can set the triple (G, x', y') to any triple as long as $\text{GCD}(x', y') = 1$.

To iterate every triple (G, x', y') such that $L \leq N$, we can iterate all tuples of three values (w_1, w_2, w_3) $(1 \leq w_1 \leq w_2 \leq w_3 \leq N)$ such that w_2 is divisible by w_1, and w_3 is divisible by w_2. Here, w_3

represents L, and we get the values:

- $G = w_1$
- $x' = \frac{w_2}{w_1}$
- $y' = \frac{w_3}{w_2}$

For each triple (G, x', y'), we can calculate the difference $(3^{\text{mul}[L]} \times 2^{\text{mul}[x]+\text{mul}[y]-2\times\text{mul}[L]} - 2^{\text{mul}[x]+\text{mul}[y]}) \times \mu(x) \times \mu(y)$ in $O(1)$.

For each value of w_2, there are $O(\frac{N}{w_2})$ different values of w_3. So for each value of w_1, there are $O(\frac{N}{w_1} \log N)$ different pairs (w_2, w_3). Therefore, there are $O(N \log^2 N)$ triples.

That's how we calculate for the total number of ways if the GCD of both decks must be k for $k = 1$. But we want to calculate for every value of k from 1 to N. Notice that for a value k, the values in A we only care about are only the multiples of k. That means, the total number of different values is $O(\frac{N}{k})$, so the total number of triples for that k is $O(\frac{N}{k} \log^2 N)$. Therefore, in total, there are $O(N \log^3 N)$ total iterations.

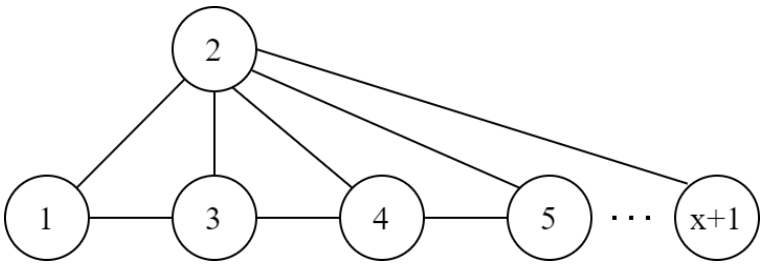
In practice, the total number of iterations is way smaller than that. We can also optimize it such that we only iterate the triples that satisfies $\text{GCD}(x', y') = 1$, $\mu(x) \neq 0$, and $\mu(y) \neq 0$. The total number of iterations is about 2×10^7.

L. Buggy DFS

Denote the return value of `BDFS()` of a graph as the k-value of the graph. Note that we can do the following to systematically increase the k-value of our graph. Formally, if we have two graphs with the k-values of x and y, we can combine the graph by merging both of their starting nodes into a single node and the k-value will be $x + y$. The proof is trivial and left as an exercise for the reader.

However, the smallest non-zero k-value is 2 (from a graph with 2 nodes connected to each other). Using the observation above, we can only create graph with an even k-value. By brute force search, it can be shown that the smallest possible odd k-value is 11. In other words, it is impossible to construct a graph with a k-value that equals to 1, 3, 5, 7, or 9. Otherwise, we can combine the graph with a k-value of 11 with any graph with an even k-value for the remaining odd k-values.

The only problem is to create a graph with at most $32\,768$ nodes and $65\,536$ edges. The following class of graph is the easiest to implement.



If the graph in this class consists of $x+1$ nodes, it will have $2x-1$ edges and k-value of $x(x+3)-2$.

We can repeatedly add zero or more graphs of this class until we get a combined k-value as close as possible to K. We also need to pay attention such that the remaining difference to the actual value K is at least 11 to make sure it's still possible to fill in the gap in case the difference is odd. To get it under the constraints, in each iteration, we repeatedly pick the largest x such that $x(x+3)-2 \leq K-11$ to make a graph of the aforementioned class to combine with the resulting graph. If at the end, it needs an odd k-value, combine our graph with the graph that has a k-value of 11. Finally, we can repeatedly combine the graph with the graph that has a k-value of 2 until our k-value is equal to K.

Let's calculate the total number of nodes and edges. Notice that from our constructing process, the number of edges is always not more than twice the number of edges. That means, we only need to make sure that the number of nodes is at most $32\,768$.

Let's say the current remaining value needed is d. Then the value of x is about $\sqrt{d}$. And then, the remaining value that's needed turns into a value near $2\sqrt{d}$ (because the difference of k-values between adjacent values of x is about $2x$). That means, from the initial value of $d = K$, we iteratively add $\sqrt{d}$ nodes to turn d into $2\sqrt{d}$, over and over again until d becomes small enough. The number of nodes added after that is constant. It can be obtained that the value of d decreases very rapidly in each iteration. Under the given constraints, it can be calculated that the total number of nodes is always not more than $32\,768$.

The time complexity of this solution is $O(\sqrt{K})$.

M. Mirror Maze

Because the grid size is small, you can solve this problem using graph traversal algorithm. To simplify the implementation, we can represent each cell as four new nodes for each of its four sides (north, south, east, west). For empty cells, we connect the north with south node, east with west node. For cells with type 1 mirror (/), we connect the north with west node, south with east node. For cells with type 2 mirror (\), then connect the north with east node, south with west node.

Also don't forget to connect the nodes for the touching sides of adjacent cells. When traversing the graph, don't forget to keep track of the mirrors that are already hit.

The time complexity of this solution is $O(NM)$.