Problem Analysis

Disclaimer: *This is an analysis of some possible ways to solve the problems of The 2022 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	Storing Eggs	Ammar Fathin Sabili
B	Magical Barrier	Wiwit Rifa'i
C	Nightmare Brother	Suhendry Effendy
D	City Hall	Muhammad Ayaz Dzulfikar
E	Substring Sort	Muhammad Ayaz Dzulfikar
F	Doubled GCD	Rafael Herman Yosef
G	The Only Mode	Suhendry Effendy
H	Grid Game	Prabowo Djonatan
I	Contingency Plan	Lie, Maximilianus Maria Kolbe
J	Sharing Bread	Prabowo Djonatan
K	Short Function	Muhammad Ayaz Dzulfikar
L	Increase the Toll Fees	Rafael Herman Yosef
M	Game Show	Ammar Fathin Sabili

Analysis Authors

Lie, Maximilianus Maria Kolbe

Prabowo Djonatan

Rafael Herman Yosef

Suhendry Effendy

A. Storing Eggs

Let $f(c, k, b_1, b_2)$ be the maximum distance between two closest eggs where the available columns are from column c to N, the remaining number of eggs is k, the egg-state of column $c-1$ is b_1, and the egg-state of column $c-2$ is b_2. The *egg-state* is a bitwise (3 bits) representing the eggs' existence on the respective column. The answer to the problem is $f(1, K, 0, 0)$.

To compute $f(c, k, b_1, b_2)$, we need to find a column d where $c \leq d \leq N$ such that putting eggs in such a column is optimal. Note that there are also $2^3 - 1$ ways of putting eggs in a column. The distance between the new eggs and the closest existing eggs is only determined by the distance to column c, the egg-state for column $c-1$ (i.e. b_1), and the egg-state for column $c-2$ (i.e b_2)—on why we need b_2 is left for the reader to figure out. Notice that b_1 is always at least 1 (as it represents the egg-state of the last column we put eggs) while b_2 can be 0.

With a dynamic programming approach, the total memory complexity is $O(NK)$, and the total time complexity is $O(N^2K)$, with all constant factors removed from the notation.

However, a plain implementation of this idea most likely will get a time limit exceeded verdict, unless done very efficiently. As you might have noticed, the discussed approach has a large constant factor, thus, any speed-up might be necessary to get accepted.

We observe 2 useful optimization/pruning: stop the state computation when there is not enough slot available to put all k eggs, stop the state computation when it cannot produce a better solution. Alternatively, we can reduce the state size by doing further analysis. Observe that the middle row state of b_2 on column $c-2$ does not matter as any egg on column $c-1$ will have a shorter distance to the new egg. Thus, we can reduce the egg-state of b_2 from 3 bits into 2 bits (only consider the first and third row).

B. Magical Barrier

In this analysis, we refer to a power source as a point and a magical barrier as a segment.

For each segment that connects point A and point B, let us instead count the number of segments that do not intersect segment $\overline{\mathrm{AB}}$.

First, for each segment $\overline{\mathrm{AB}}$, we will try to separate other points according to the line $\overline{\mathrm{AB}}$. Let $\mathrm{CCW[A, B]}$ be all P such that the vector $\overrightarrow{AP}$ is in the counter-clockwise direction from $\overrightarrow{AB}$. More formally, $\mathrm{CCW[A, B]} := \{\mathrm{P} : (\overrightarrow{AB} \times \overrightarrow{AP}) \cdot (0, 0, 1) > 0\}$. Then, each point P (other than A and B) is part of either $\mathrm{CCW[A, B]}$ or $\mathrm{CCW[B, A]}$. All the values of $\mathrm{CCW[A, B]}$ can be computed in $O(N^2 \log N)$ using radial line sweep.

Segments that neither touch nor intersect segment $\overline{\mathrm{AB}}$ can be categorized into four types:

1. both points connected by the segment is part of $\mathrm{CCW[A, B]}$;
2. both points connected by the segment is part of $\mathrm{CCW[B, A]}$;
3. the segment intersects ray $\overrightarrow{\mathrm{AB}}$ but not segment $\overline{\mathrm{AB}}$; and
4. the segment intersects ray $\overrightarrow{\mathrm{BA}}$ but not segment $\overline{\mathrm{AB}}$.

The number of segments of type 1 and 4 can be obtained by computing:

$$S(A, B) := \sum_{P \in \mathrm{CCW[A,B]}} |\mathrm{CCW[A, P]}|$$

while the number of segments of type 2 and 3 is:

$$S(B, A) := \sum_{P \in \mathrm{CCW[B,A]}} |\mathrm{CCW[B, P]}|$$

For each point A, computing $S(A, B)$ for all point B can be done in $O(N \log N)$ using radial line sweep. Hence, all the values of $S(A, B)$ can be computed in $O(N^2 \log N)$.

Once all the calculations above are done, we can find the number of segments that intersects $\overline{\mathrm{AB}}$ by computing $\frac{(N-2)(N-3)}{2} - S(A, B) - S(B, A)$. Thus, we only need to find the maximum value for all A and B.

C. Nightmare Brother

As the constraints are quite small, we can simply simulate for each hint i to find what the resulting string will be if hint i is false. Constructing such a resulting string for each simulation can be done in $O(NM)$. There are many ways to maintain the number of unique outputs, e,g., with `std::set` container. Also, don't forget the case when there is no false hint.

D. City Hall

Let $adj(i)$ be the set of intersections that are directly connected to intersection i. Also, let $distS_i$ be the minimum amount of energy needed to go from intersection S to intersection i, and $distT_i$ be the minimum amount of energy needed to go from intersection T to intersection i. The single source shortest path algorithm can be used to compute $distS_i$ and $distT_i$ for all i starting from intersection S and T, respectively.

There are 3 possible scenarios.

1. The altitude to be changed is at intersection S
2. The altitude to be changed is at intersection T
3. The altitude to be changed is at intersection i where $i \neq S$ and $i \neq T$

In the 1^{st} scenario, we simply change H_S into one of H_i where $i \in adj(S)$ so that the energy required to traverse the road connecting S and its direct neighbor is 0. The minimum amount of energy needed in this case is equal to $distT_j$ where $j \in adj(S)$. Using a similar approach, we can find the minimum amount of energy for the 2^{nd} scenario as well.

As for the 3^{rd} scenario, let $f(i)$ be the minimum amount of energy from S to T if we change H_i. The following formula represents $f(i)$,

$$f(i) = \min_{u,v \in adj(i)} \{distS_u + distT_v + d(u,v)\}$$

$$d(u,v) = (H_u - opt(u,v))^2 + (H_v - opt(u,v))^2$$

where $opt(u,v)$ represents the new altitude to minimize $d(u,v)$. It turns out that the optimal $opt(u,v)$ is equal to $\frac{H(u)+H(v)}{2}$ (proof omitted).

Substituting $opt(u,v)$ with the optimal one, we can rewrite $f(i)$ into

$$f(i) = \min_{u,v \in adj(i)} \left\{ distS_u + distT_v + \frac{H_u^2}{2} + \frac{H_v^2}{2} - H_u \cdot H_v \right\}$$

Solving this formula for each i with a direct approach (testing all pairs $u, v \in adj(i)$) will get you a time limit exceeded verdict as it has a time complexity of $O(M^2)$. By rearranging the formula into

$$f(i) = \min_{u \in adj(i)} \left\{ distS_u + \frac{H_u^2}{2} + \min_{v \in adj(i)} \left\{ distT_v + \frac{H_v^2}{2} - H_u \cdot H_v \right\} \right\}$$

, we can see that $distT_v + \frac{H_v^2}{2} - H_u \cdot H_v$ is a linear equation that depends on H_u. Therefore, we can utilize the convex hull trick or Li Chao Tree to find the optimum i and its new H_i value. The time complexity for the 3^{rd} scenario depends on what trick you employ, e.g., $O(M \log M)$ with the convex hull trick.

E. Substring Sort

This problem can be solved with the square root decomposition technique. For the sake of simplicity, assume N is a square number (i.e. $\sqrt{N}$ is an integer), and let k be $\sqrt{N}$.

First, we need to create k partitions (blocks) for each string A, B, and C where the length of each string in each block is also k. Let a be the partitions for A. The notation a_i represents the i^{th} block of A that contains the string $A_{(i-1)\cdot k+1}...A_{i\cdot k}$. Similarly, b for B and c for C.

The main idea is to represent each block (of size k) with a single integer that can easily be compared against other strings at the same block index. Computing, comparing, and maintaining these integers can be done in $O(\sqrt{N})$ for each index i.

For each index i, let ra_i, rb_i, and rc_i be the rank of a_i, b_i, and c_i, respectively. For example, let $a_i =$ `MMM`, $b_i =$ `DDD`, and $c_i =$ `RRR`; the sorted order is $b_i < a_i < c_i$, thus, $ra_i = 2$, $rb_i = 1$, and $rc_i = 3$. These values can be computed in $O(\sqrt{N})$ for each index i.

For each query $\langle l, r\rangle$, we aim to represent the substring of length $O(N)$ with a shorter object (we'll use a vector of integers) of length $O(\sqrt{N})$. The intended substring in each query contains at most 3 parts with respect to the partitions: the head (zero or more characters), the body (zero or more blocks), and the tail (zero or more characters). For the body, we can represent each block with its rank. For each character in the head and tail, we can simply use the characters themselves (represent each character with an integer, e.g., `'z'` $= 25$). As the size of each of those 3 parts is $O(\sqrt{N})$, the copy operation can be done in $O(\sqrt{N})$.

Sorting 3 items of $O(\sqrt{N})$ in size can be done in $O(\sqrt{N})$.

Then, we need to maintain each a_i, b_i, c_i, ra_i, rb_i, and rc_i according to their new order in the replace operation (perform swaps where necessary). For the head and tail parts, we can swap the characters between strings manually. For each query, there will be at most two blocks for each string that are partially updated (the head and the tail); we need to recompute the rank for this block index. Therefore, the replace operation can be done in $O(\sqrt{N})$.

This idea solves all the procedures in $O(\sqrt{N})$ per query, thus, the total time complexity is $O(Q\sqrt{N})$.

F. Doubled GCD

Performing all $N-1$ moves is similar to finding the greatest common divisor (GCD) of all A_i; however, there might be additional factors of 2 due to the moves (doubled the GCD).

First, let's extract $GCD(A)$ from A_i. Let B_i be $A_i/GCD(A)$, and let C_i be the number of factors 2 in B_i, e.g., $B_i = 40 = 2^3 \times 5 \rightarrow C_i = 3$. Choosing two cards i and j will cause the number of factors 2 in the resulting new card to be $\min(C_i + C_j) + 1$. Therefore, to get the maximum possible number of factors 2 in the last card, we can greedily pair two indices i and j that have the lowest number of factors 2, i.e. C_i and C_j are the smallest among C. Let the number of factors 2 in the

last card be P, then the output is $GCD(A) \times P$.

G. The Only Mode

In this analysis, we only consider the part where the pivot $x = 0$; any other x can be done similarly.

Firstly, construct a partial sum $p_k(m)$ thas represents the number of occurences of k in $A_{1..m}$ for each $k = \{0, 1, 2, 3\}$. Then, let $f_k(m)$ be $p_0(m) - p_k(m)$ for $k = \{1, 2, 3\}$. Notice that $f_k(m)$ will be positive if there are more 0 in $A_{1..m}$ than k, negative if it's the other way around, and zero if they're equal. As $f_k(m)$ is also a partial sum, then we can find such a sum on a specific range $[L, R]$, i.e. $A_{L..R}$, with $f_k(R) - f_k(L - 1)$.

To find a range where 0 appears stricly more often than any other k, we have to find a range $[L, R]$ where $f_k(R) - f_k(L - 1) > 0$ for all $k = \{1, 2, 3\}$. This can be done with the help of a geometric data structure that supports point updates and range queries.

Represent each tuple $\langle f_1(m), f_2(m), f_3(m)\rangle$ as a point in a 3-dimensional plane. Process each point one by one from $R = 1$ to N. For each index R, we need to find a point $\langle f_1(L-1), f_2(L-1), f_3(L-1)\rangle$ that are strictly on the left (smaller on all dimensions) of $\langle f_1(R), f_2(R), f_3(R)\rangle$ where $L = 1..R - 1$.

But, hold on! Unless there is a 3-dimensional data structure that can handle those queries efficiently enough, we still need to speed up this solution.

Luckily, we can discard one of the dimensions. Instead of processing R from 1 to N, we can process R from i such that $f_1(i)$ is the smallest until the largest. By doing so, we don't need to consider $f_1(m)$ anymore in the search as the points that have been processed do not have a larger $f_1(m)$; though, we still need to be careful when $f_1(m)$ is equal. Therefore, we only need a 2-dimensional data structure, e.g., a 2d segment tree, or a simpler 2d BIT (since all the queries are half-plane in each dimension).

H. Grid Game

Let's call a cell to be *dead-end* if the cell is passable and both the right side and bottom side of the cell are impassable (or outside of the grid).

Find the xor of all the integers written on the dead-end cells. If the resulting xor is a positive integer, the first player will win. Otherwise, the second player will win.

Let's prove the statements above.

Any path taken by a player starting from any position will always end on a dead-end cell.

Proof. Suppose the player doesn't end his turn on a dead-end cell. That means there still exists a passable cell that is either to the right or the bottom of the end's cell, hence the player must still make a move, a contradiction. □

Moreover, the last cell passed by a player on his turn is the only dead-end cell that is passed by the player on that turn.

Proof. Suppose there exists another cell on the path that is a dead-end cell, then that player couldn't make any more moves on that cell because neither the right nor the bottom side of the cell is passable, a contradiction. □

Also, notice that each integer on the path passed by a player on his turn is always replaced by a different integer. This is because the integer x at the starting cell and the chosen integer y are different, hence $x \oplus y$ is a positive integer, and any integer that is xor-ed with a positive integer will yield a different value.

Therefore, the losing player will always replace an integer from exactly one dead-end cell with a different integer. The winning player can treat the integers on the dead-end cells as a standard nim game, and apply the same winning nim game strategy by starting his move on the dead-end cells.

I. Contingency Plan

There are several solutions that can construct a valid contingency plan. We discuss one of the solutions.

First, any star graph does not have a valid plan. The following construction algorithm shows that all trees other than a star graph have a solution.

Start by choosing an arbitrary node as the root of the tree; let node r be the root. Let $C_i(r)$ be the i^{th} child of r, sorted by the edge number in the given tree. For all nodes other than r and its children, it can be connected to r immediately. Then, for the children of r, $C_i(r)$ can be connected to $C_{i+1}(r)$, except for the last child of r; let c be this node. Finally, for node c, connect it to any node that satisfies all the following requirements.

- It cannot be r and the children of c
- It cannot be the other children of r
- It must already be connected by a backup cable

If such a node exists, then connect c to that node. If such a node does not exist, then the root must be changed. Note that selecting the children of r as the new root might result in the same problem. However, any children of c can be selected as the new root, and it is guaranteed for this new root to have a valid plan (it can be proven with contradiction).

This construction also implies that if c has any child, then a valid plan must exist. The other cases are either when the original root r already has a valid plan, or the tree is a star graph that has no valid plan.

J. Sharing Bread

Consider that we have $N+1$ toasters instead, and they are arranged in a circular manner (i.e. when a person looking at toaster $N+1$ has no bread, the person will continue to search from toaster 1). After all M people have taken the bread, there will be exactly $N-M+1$ toasters left with bread.

The answer for the original problem will be equivalent to the number of ways to take the bread on the circular-arranged toaster and toaster $N+1$ is left with bread. Now, the number of starting sequences in this circular setting is $(N+1)^M$. Out of those, $\frac{N-M+1}{N+1}$ of them left toaster $N+1$ with bread. The ratio is proportional to the number of bread left because there exists a way to partition all the starting sequences into classes such that each class has $N-M+1$ starting sequences that left toaster $N+1$ with bread and M starting sequences that left toaster $N+1$ with no bread. One such partitioning is for $(a_1+i, a_2+i, \ldots, a_M+i)$ for all $1 \le i \le N+1$ belonging to the same class.

Therefore, the final answer is $(N-M+1)(N+1)^{M-1}$.

K. Short Function

First, observe that the result for B_i is $(A_i \times A_{i+1} \times ... \times A_{(i+2^K-1) \bmod N}) \bmod M$ where M is a prime number $998\,244\,353$. We can split this answer into two components.

1. The repeating part: $((\prod A_i)^{\lfloor 2^K/N \rfloor}) \bmod M$
2. The tail: $(A_i \times A_{i+1} \times ... \times A_{(i+(2^K \bmod N)-1) \bmod N}) \bmod M$

The first part will be the same for all i, thus, we only need to compute it once. It can be calculated by utilizing Fermat Little Theorem so the first part becomes $((\prod A_i)^{\lfloor 2^K/N \rfloor \bmod (M-1)}) \bmod M$. On the other hand, the second part can be calculated with the help of prefix/suffix multiplication. Finally, to get the answer for B_i, simply multiply both parts and modulo with M.

Another challenge in this problem is how to compute $\lfloor 2^K/N \rfloor \bmod (M-1)$. One way to achieve this is by representing 2 with a pair $\langle p, q \rangle$ where $2 = p \times N + q \bmod N$. This way, we split the quotient and the remainder when divided by N. Observe that to perform a floor operation on $\langle p, q \rangle$, we only need to take the quotient p. Then, to get $2^K \bmod (M-1)$, simply do modular binary exponentiation on $\langle p, q \rangle$ with K as its power. When we multiply $\langle p_1, q_1 \rangle$ with $\langle p_2, q_2 \rangle$, don't forget to maintain the quotient/remainder pair with respect to N. Let the result be $\langle p', q' \rangle$, then $\lfloor 2^K/N \rfloor \bmod (M-1)$ is equal to p'.

L. Increase the Toll Fees

First, we can use MST (Minimum Spanning Tree) algorithm to determine the set of toll roads that have been used before the price increase. To check if the King's plan is impossible, we can do MST algorithm for the second time without including the toll roads from the first MST. If there are some cities that are not connected after executing MST algorithm for the second time, the result is -1.

The minimum total increase can only be obtained if we only increase the fees from the set of toll roads from the first MST. For toll road i that connects city U_i to V_i in the first MST, we can change its fee with $d(U_i, V_i) + 1$ where $d(u, v)$ represents the maximum toll fee from path u to v in the second MST.

To find the $d(u, v)$ efficiently, we can use the lowest common ancestor (LCA) algorithm. Hence, this problem can be solved in $O(NlogN)$.

M. Game Show

The round is flawed if there is a cycle (starts and ends at region i) such that the penalty is negative. For each round, the contestant can go to region i and use this cycle to reduce the penalty infinitely many times.

To check if there is a cycle with negative penalty, at least one of these conditions must be satisfied.

1. Sum of all A_i is negative
2. Sum of all B_i is negative
3. There exists a region i such that A_i + B_i is negative

If $A_{x..y} + B_{x..y} < 0$, then there is a region k such that $A_k + B_k$ is negative. Therefore, checking if there exists an integer i such that $A_i + B_i < 0$ is sufficient.

If the round is not flawed, then the minimum penalty from region s to t can be obtained simply by using path A only, i.e. $A_s + A_{s+1} + ... + A_{t-1}$ or path B only, i.e. $B_{s-1} + B_{s-2} + ... + B_t$. But a direct implementation will get a time limit exceeded verdict. We can speed up the process by using the prefix sum for A and B. Hence, we can calculate the penalty of each round in $O(1)$. Also, don't forget to handle the circular setting.