Problem Analysis

Problem Title		**Problem Author**	**Analysis Author**
A	Edit Distance	Alham Fikri Aji, Jonathan Irvin G.	Suhendry Effendy
B	Rotating Gears	Muhammad Rais Fathin Mudzakir	Jonathan Irvin Gunawan
C	Smart Thief	Suhendry Effendy	Suhendry Effendy
D	Icy Land	Jonathan Irvin Gunawan	Suhendry Effendy
E	Artilleries and Defensive Walls	Vincentius Madya Putra I.	Suhendry Effendy
F	Popping Balloon	Alham Fikri Aji	Jonathan Irvin Gunawan
G	Go and Make It Complete	Suhendry Effendy	Suhendry Effendy
H	Lexical Sign Sequence	Alfonsus Raditya A., Jonathan I. G.	Jonathan Irvin Gunawan
I	Lie Detector	Jonathan Irvin Gunawan	Suhendry Effendy
J	Future Generation	Ashar Fuadi	Ashar Fuadi, Jonathan I. G.
K	Boomerangs	Ivan Adrian Koswara	Jonathan Irvin Gunawan
L	Binary String	Jonathan Irvin Gunawan	Suhendry Effendy

A. Edit Distance

If the problem requires us to find a binary string T of the same length as the input S which has the largest edit distance to S, then this problem can be more challenging. Fortunately, that's not the case; this problem only asks us to find a binary string T of the same length as input which has an edit distance strictly larger than half of S's length.

There are many possible outputs which satisfy the requirement. Here, we will discuss a simple one.

First, count how many `0` and `1` does S has. If S has more `0` than `1`, then simply set T to be all `1`. If S has more `1` than `0`, then simply set T to be all `0`. The problem is what we should do with the case where the number of `0` and `1` are equal. In this case, we can set T to be either `01111..` or `10000..` depends on the first character of S, i.e. S_1. Specifically, if S_1 is `0`, then T is `10000..`; otherwise, T is `01111..`.

B. Rotating Gears

Let gear 1 be the root of our tree. The first observation to solve this problem is that we can simplify the problem such that all the gears connected to the gear given in the type 3 operation rotate in the same direction. However, for each $3\ x\ \alpha$ operation, we must negate the value of α if the distance (number of edges) between gear 1 and gear x is odd. Moreover, after all the operations, we must negate the value of δ_x for all gear x that has an odd distance to gear 1.

The main idea to solve this problem is for each operation $3\ x\ \alpha$, we find the highest ancestor of gear x that is not taken out and still connected to gear x. We can do this efficiently using a binary search, and, for each gear g, we store the number of taken out gears that is an ancestor of g. Let's say the highest ancestor of gear x that is still connected to gear x is gear y. We then increment δ_z by α, for all gear z in the subtree of gear y.

This is not necessarily true, since there might be a gear u in the subtree which is not connected to gear y, and we don't want to update the value of δ_u. Let gear v be the lowest ancestor of gear u which is taken out (not on the board). Since gear u is not connected to gear y, gear v must be in the path between gear u and gear y. The idea is to note the value of δ_v when it was about to be taken out. Let this value be δ'_v. When we place gear v back to the board, we can correct the value of δ_w for all gear w in the subtree of gear v by incrementing $\delta'_v - \delta_v$. The exact details of the increment values when there are two gears taken out, where one gear is the ancestor of the other, is left out in this discussion.

After all the operations are done, we can simply return all the taken out gears to the board to make sure that the δ value correctness takes place, without changing the output. In other words, we can assume that there are additional pseudo-operations that return all the gears back.

The number of gears connected to gear x can be computed using the same idea. For each of the taken out gear (say, gear g), we compute the number of gears in the subtree of gear g that is still connected to gear g. Let this value be μ_g.

The number of gears connected to gear x can be retrieved by $subtree(y) - \sum \mu_z$, for all gear z in the subtree of y, and $subtree(y)$ is the number of gears in the subtree of gear y. The details of updating the value of μ when gears are taken out or placed back are also left out in this discussion.

All the data structure needs (incrementing all numbers in a subtree for δ, finding the highest ancestor that is still connected to the current node, and finding a sum of numbers in a subtree for μ) requires either a range-update point-query or a point-update range-query data structure. We can use Fenwick Tree for all of them. The solution runs in $O(N \log^2 N)$.

C. Smart Thief

Observe that the length of the shortest string which contains K distinct substrings of length N (in other words, the length of the output) is exactly $N + K - 1$. More specifically, the output for this problem corresponds to a partial *de Bruijn sequence*.

Let $G = (V, E)$ be a directed graph where the set of nodes V be all possible strings of length $N - 1$, and a node u has a directed edge (u, v) to node v if and only if the string represented by node v can be obtained by appending one character to u and deleting the front-most character of u, e.g., “`abcd`” $\rightarrow$ “`bcde`” and the corresponding directed edge’s label is “`e`”. Then, the output for this problem corresponds to an *Eulerian path* of length K in G.

However, the constraint for N in the problem can be as large as 100000, thus, we cannot simply build G as described. Note that there are $M^{(N-1)}$ nodes in G.

Let’s consider two separated cases.

1. Small N (e.g., ≤ 40). In this case, we can solve the problem by using graph G, however, we should not construct G as it. Instead, we should construct G *on-the-fly* (as needed) and stop once we obtained the desired result, thus, we only have a “partial” G. With *Hierholzer’s algorithm* to find an Eulerian path, we can solve this in $O(NK)$ time, or $O(NK \log N)$ if you’re using an easier implementation.
2. Large N. In this case, simply output a random string (with the given alphabets) of length $N + K - 1$. Note that there are M^N distinct substrings of length N. If N is large and $M \geq 2$, then $M^N \gg K$, thus, the probability that the same substring appear more than once in the output is extremely low. Additionally, you can write a function to check whether the output string is valid, e.g., with *rolling hash* function (like the one implemented in the checker program for this problem), but it’s not necessary.

A note on Hierholzer’s algorithm. This algorithm finds an Eulerian tour in $O(E)$ and works by maintaining two lists, i.e. exploring list and result list. At any time, each list contains a valid Eulerian path. So, for the purpose of this problem, we can simply terminate when any of the lists contains $\geq K$ nodes, and work the output from that list.

Also, note that there are some implementation details not mentioned in this analysis for you to figure out.

D. Icy Land

First, let us consider the case where $R > 2$ and $C > 2$. In this case, all the *inner* cell, i.e. $r = [2, R - 1]$ and $c = [2, C - 1]$, should be a dry land as there will be no way to stop on the inner cell if it is an icy land. On the first row, last row, first column, and last column, there should be at least one dry land which does not lie on the corner. This dry land is used for us to enter the inner cells.

For example, in the following board, we can visit all cells regardless of the starting point.

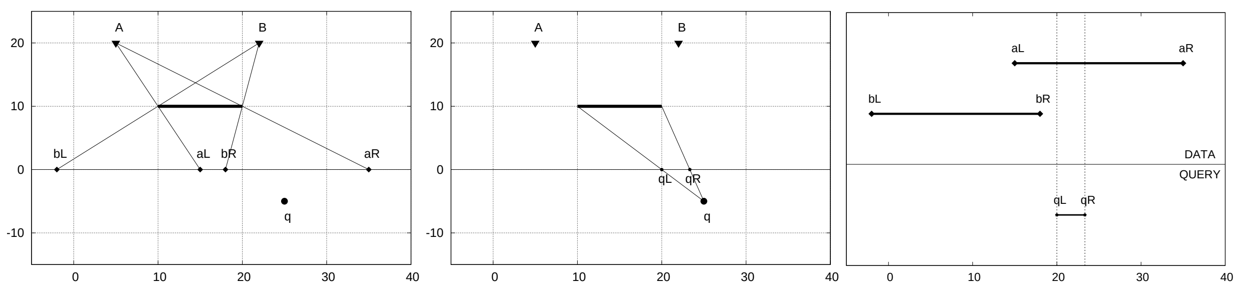
```
.....#.
.#####.
.######
.#####.
######.
.#####.
...#...
```

For the case where $R \leq 2$, we only have to make sure that there is at least one dry land in each column $c = [2, C-1]$. Similarly, in the case where $C \leq 2$, we only have to make sure that there is at least one dry land in each row $r = [2, R-1]$.

E. Artilleries and Defensive Walls

A naïve solution to answer each query q would be simply iterating through each artillery point $p \in P$ and check whether the line connecting q and p intersect with any wall. This solution has a complexity of $O(NM)$ per query, and for this problem, this solution will get you `TIMELIMIT` as the number of artilleries and queries may reach 40000 each. We need to do better.

First, let's consider the case when $M = 1$ ($M = 0$ is trivial and not interesting). We can answer each query in $O(\log N)$ time with a preprocess step (for all queries) of $O(N \log N)$. For each point p which lies above the wall, project the wall from point of view p to $y = 0$. Let the projection be p_L and p_R. Let S be the collection of all $\langle p_L, p_R \rangle$. Now, to answer a query q, do a similar thing, i.e. project the wall from q's point of view to $y = 0$. Let the projection be q_L and q_R. The number of artilleries which cannot be seen from q will be equal to the number of tuples $\langle p_L, p_R \rangle$ in S such that $p_L \leq q_L \leq q_R \leq p_R$. You can use any 2-dimensional data structure to answer such query; however, as the data are static (no alteration), a much simpler solution like binary search is better (you also want to avoid a large constant factor in your solution).



Now, to handle $M > 2$, we can use the *inclusion-exclusion principle*. Let $\langle p_L^m, p_R^m \rangle$ be the intersection projection of set of wall m from p's point of view to $y = 0$. In other words, any line from any position $(x, 0)$ where $x = [p_L^m, p_R^m]$ to p will intersect with all m. Calculate for all possible set of wall m, and the answer to each query then can be computed simply by the inclusion-exclusion principle. The preparation step (one-time) for this approach is $O(2^M N(M + \log(NM)))$, and the query processing is $O(2^M(M + \log(NM)))$ per query.

F. Popping Balloon

We can easily compute the number of tasks that Ayu is going to solve in the contest time. Let S be this number. Therefore, now Ayu's goal is to make sure that Budi did not solve the S^{th} problem.

Observe that Ayu can always pop a balloon just before Budi about to solve a task, since if Budi is going to solve the next task in k minutes, it is always not worse to let Budi waste the full k minutes. Therefore, Ayu can decide whether she wants to pop a balloon only every time Budi just about to solve a task.

Suppose Budi is just about to solve a task (let's say the x^{th} task), how does Ayu determine whether she should pop a balloon? First, if Ayu currently does not have a balloon, then there is no choice but to let Budi solve the task. Otherwise, consider the tasks between task x and task S, inclusive. If task x takes the longest to solve for Budi, then Ayu should spend as many balloons as she can on this task. This means Budi will waste as many minutes as Ayu can. If there is a later task (let's say the y^{th} task) that Budi needs more time to solve, then it is better for Ayu to save her balloon and use it for the y^{th} task instead. This ensures that Budi wastes more time.

G. Go and Make It Complete

One way to solve this problem is by using binary search on the output k and test whether $go(G, k)$ can produce a complete graph. Whether $go(G, k)$ can produce a complete graph can be tested in $O(N^3)$.

Maintain a list Q containing candidate edges (a, b) where $\delta_a + \delta_b \geq k$. Process (add) edge $q \in Q$ into G in any order. When (a, b) is added, scan through all potential neighbours c of a and add (a, c) into Q if (a, c) is not in G or Q and $\delta_c + \delta_c \geq k$ (be careful with duplicate entries). Do the same for node b. Process all edges in Q. Note that if $go(G, k)$ can produce a complete graph, then this method will produce a complete graph. This algorithm runs in $O(N^3)$.

Combined with the binary search, the time-complexity for this solution is $O(N^3 \log N)$. Be careful when you decide the binary search range for k. The largest possible output is $(N - 2) + (N - 2)$, not $N - 1$.

You can use other approaches with asymptotically the same time-complexity, but beware with the hidden constant factor in your solution. If you didn't implement your solution efficiently, your solution might get `TIMELIMIT` even though it has a theoretical $O(N^3 \log N)$ time-complexity—as what happened when the judges prepared this problem, e.g., < 1 second vs 30 seconds.

H. Lexical Sign Sequence

We can convert each of the constraints A_i B_i C_i into the following: there are at most $D_i = \frac{B_i - A_i + 1 - C_i}{2}$ -1s in the range $[A_i, B_i]$. We can imagine D_i as the “hit points” (HP) of the i^{th} constraint. Whenever we put a -1 in $[A_i, B_i]$, we decrement the HP of the i^{th} constraint by one.

We can also subtract D_i with the number of $-1s$ already prefilled in the range $[A_i, B_i]$, thus we can ignore the prefilled conditions for the rest of the discussion. If one of D_i is less than 0, then it is impossible to fill the sequence.

Since we want to find the lexicographically smallest sequence, we want to fill the sequence greedily from the first position. As long as we can fill a position with -1, we will do so. We cannot fill a position with -1 if doing so causes one of the constraint to have a negative HP. However, an $O(NM)$ solution by checking each constraint for each position is too slow for this problem.

We can maintain a data structure to keep track of the constraint with the least HP. At the start of our loop at position x, we can insert those constraints i with $A_i = x$ to our data structure. Also, at the end of our loop at position x, we can remove those constraints i with $B_i = x$. We check whether it is possible to put a -1 in the position x by checking the least value in our data structure. If and only if it is larger than 0, then can put a -1 in the position x. If we decide to put a -1 in the position x, we need to subtract all values in our data structure by one. We can implement this by actually inserting $D_i + C$ (instead of D_i) to our data structure, where C is the number of -1s we have put so far. This is to make sure that all the constraints HP in the data structure are “calibrated” using the same benchmark.

To support our needs, we can either use a balanced binary search tree or a minimum heap with lazy deletion. This algorithm runs in $O(N \log N)$.

I. Lie Detector

This is the easiest problem in the 2018 ICPC Asia Jakarta Regional Contest. All you have to do is to check whether there is an **odd** number of `LIE`. If yes, then the output is `LIE`, otherwise, `TRUTH`.

J. Future Generation

Let M be the number of characters in the longest name given by Andi’s soon-to-be-grandmother-in-law (i.e. $M = \max(|S_i|)$.

This problem can be solved using a dynamic programming. Let $dp(i, mask)$ be the maximum possible total length of children names if we only consider $S_1, S_2, \ldots, S_i$ and we take the subsequence of S_i as denoted by $mask$.

The recurrence is $dp(i, mask) = popcount(mask) + \min(dp(i-1, prev))$, where $prev$ are masks in the range $[0, 2^{|S_{i-1}|})$ that satisfies the subsequence of S_i as denoted by $mask$ is lexicographically larger than the subsequence of S_{i-1} as denoted by $prev$, and $popcount(mask)$ is the number of active bits in $mask$. However, this DP runs in $O(N \times M \times 2^M \times 2^M)$ (since it has $O(N \times 2^M)$ states and we need $O(M \times 2^M)$ time for the recurrence). This is too slow.

Let us improve this DP. First, we precompute all subsequences of each name and then sort them lexicographically with their corresponding mask. Then, we can find the largest $prev$ of S_{i-1} that is still "less" than $mask$ of S_i (here "less" means that the subsequence of S_i as denoted by $mask$ is lexicographically larger than the subsequence of S_{i-1} as denoted by $prev$), using binary search. We also store the prefix minimum of the previous DP row.

Now the recurrence roughly becomes $dp(i, mask) = popcount(mask) + dpMin(i-1, prev)$, where $prev$ is found by binary search. This DP runs in $O(N \times M^2 \times 2^M)$.

We can optimize this further by computing all values of $dp(i)$ using two-pointers technique from $dp(i-1)$, since the value of $prev$ we want from $dp(i, mask)$ to $dp(i, mask+1)$ never decreases. This optimizes the DP to run in $O(N \times M \times 2^M)$.

K. Boomerangs

If the graph is not connected, we can run the boomerang partition algorithm independently for each of the connected components. Therefore, the rest of the discussion assumes that the graph is connected.

For now, assume that the graph has even number of edges. Since the boomerangs must be disjoint, and each boomerang uses two edges, there are at most $\frac{|E|}{2}$ boomerangs that we can find. It turns out that we can find $\frac{|E|}{2}$ boomerangs (i.e. a partition of boomerangs), thus finding the optimal number of boomerangs.

We can prove this by induction. If we have a connected graph with 2 edges, we can obviously find a partition of boomerangs. Also, we will prove that we can find a partition of boomerangs on a connected graph with x (even) edges if we can find a partition of boomerangs on a connected graph with y (even) edges for all $y < x$.

Suppose we have a connected graph G with an even number of edges. There are two cases:

- Suppose there is no bridge in G. We can always find a boomerang in G where removing the boomerang edges in G causes either G to remain connected, or to be separated into two connected components, each having an even number of edges.
- Suppose there is a bridge (u, v) in G. Let U and V be the two connected components if we remove the edge u, v, with u in U and v in V. Since G has even number of edges, exactly one of U or V must have an even number of edges. Without loss of generality, suppose U has an even number of edges. We can choose a node w in V such that removing the edge (v, w) causes V to remain a connected component, or V to be separated into two connected components, each having an even number of edges.

In both cases, we can find one boomerang, and removing them from G may cause G to be separated into several connected components. However, since we can find a partition of boomerangs in each of the connected components, we can find a partition of boomerangs in G.

Implementing the induction above using recursion is tricky—we need a data structure that returns a bridge in an online manner. Fortunately, there is a simpler algorithm. To make the discussion easier, we will first find a partition of boomerangs on a tree with an even number of edges.

Let T be a tree with an even number of edges. While there is an edge on T, we do the following:

- Suppose there is a node u having two children v and w which are leaves. We can create a boomerang (v, u, w) and remove these edges.
- Otherwise, we can always find a node u with only one child, and that one child is a leaf. We can create a boomerang $(child(u), u, parent(u))$ and remove these edges.

Since the rest of the tree is still connected, we can still find a partition of boomerangs. We can implement this in $O(N)$ time using DFS.

We can convert the original graph G into a tree while keeping the number of edges by adding the edge in G one by one. If adding an edge (u, v) creates a cycle, we add an edge (u, w) instead, where w is a new node. However, when we print the solutions, keep in mind that w's original ID is actually v. This algorithm runs in $O(N \log N)$.

The last piece of the puzzle is to figure out what to do if we have an odd number of edges. We can simply discard any one of them, since an additional one edge is not enough to create another boomerang.

L. Binary String

At first glance, it looks like this problem can be solved by a greedy approach. Indeed, greedy works for this problem.

First, we should compute the length of the given K in binary representation. Let's say the length is $len(K)$. If $len(K) < |S|$, then simply output 0 as K is already no larger than the decimal representation of S. The *real* problem is when $len(K) \geq |S|$.

If $len(K) > |S|$, we need to remove bits from K such that its length in binary representation is at most $|S|$. The question is, which bit to remove? Our objective is to make K as low as possible by removing a minimum number of bits. Thus, we should prioritize the **second** most significant `1` bit because it's the second bit which contributes the most to the value of K. Note that the most significant `1` bit (the first one) cannot be removed as the problem states that the resulting binary string must not contain any leading zeroes. Perform this action repeatedly until $len(K) = |S|$ or there is no more `1` bit (except the most significant one). If there is no more `1` bit to be removed while $len(K) > |S|$, then remove any `0` bit to make $len(K) = |S|$. Once we get $len(K) = |S|$ with this method, we still need one additional check.

If $len(K) = |S|$ but K is still larger than the decimal representation of S, then simply remove any one `0` bit from K, and we're done.