Problem Analysis

Disclaimer: *This is an analysis of some possible ways to solve the problems of The 2023 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	Easy As ABC	Ammar Fathin Sabili
B	Button Pressing	Prabowo Djonatan
C	Cursed Game	Ammar Fathin Sabili
D	Space Exploration	Rafael Herman Yosef
E	Merge Not Sort	Ammar Fathin Sabili
F	Maximize The Value	Rafael Herman Yosef
G	Grid Game 2	Prabowo Djonatan
H	Twin Friends	Ashar Fuadi
I	Contingency Plan 2	Muhammad Ayaz Dzulfikar
J	Count BFS Graph	Muhammad Ayaz Dzulfikar
K	Deck-Building Game	Muhammad Ayaz Dzulfikar
L	Palindromic Parentheses	Lie, Maximilianus Maria Kolbe
M	Triangle Construction	Ammar Fathin Sabili

Analysis Authors

Ammar Fathin Sabili

Ashar Fuadi

Lie, Maximilianus Maria Kolbe

Muhammad Ayaz Dzulfikar

Prabowo Djonatan

Rafael Herman Yosef

A. Easy As ABC

This is a bonus task, and here is one possible solution.

First, initialize an empty vector V. Then, use a 6-nested loop to iterate through r_1, c_1, r_2, c_2, r_3, c_3; representing all possible three cells coordinates (r_1, c_1), (r_2, c_2), and (r_3, c_3). Check if all cells are three distinct cells, the first cell is adjacent to the second cell, and the second cell is adjacent to the third cell. Insert the word created by the three cells into vector V if all checkings are satisfied. Finally, find the minimum element in vector V.

Let N be the number of rows or columns in the input. Therefore, the total time complexity will be $O(N^6)$, which is fast enough given that $N = 3$.

The total time complexity for this problem is $O(N^6)$.

B. Button Pressing

There are multiple solutions to this problem which ultimately boil down to finding invariants or equivalent classes of the lamp states after button presses. Here, we present one of the solutions.

Let the current state of the lamps be $a_1, \ldots, a_N$. Consider the *prefix xor* of the lamp states, i.e. define $p_i := p_{i-1} \oplus a_i$ (for $1 \leq i \leq N$), and we set $p_0 := 0$ for now.

Observe that pressing button i is equivalent to swapping adjacent elements p_{i-1} and p_i. To see why, we observe the following two behaviours.

- Since it toggles the state of lamp $i - 1$ and lamp $i + 1$, then it will toggle both p_{i-1} and p_i.
- Since the pressed button is for a lamp that is on, that means p_{i-1} and p_i are different before (and after) the press.

Notice that p_0 is involved in the operation, hence we have to consider the dual prefixes of lamp states: one that uses $p_0 = 0$ and the other that uses $p_0 = 1$ (which flips all the remaining bits in the prefix).

Two prefixes are "reachable" within each other if both of them have the same number of active bits.

To summarize, let c_a and c_b be the number of 1 in the prefix xor of A and B respectively, we simply have to check whether $c_a \in \{c_b, N + 1 - c_b\}$.

The total time complexity for this problem is $O(N)$.

C. Cursed Game

For $N \geq 5$, there is a simple strategy to win the round in only 2 queries. For the first query, just blacken the cell $(3, 3)$ and let the rest be white. There will be a hole at cell (x, y) of the secret paper iff the cell $(4 - x, 4 - y)$ of the result grid is 1. After knowing the secret paper, it is just a matter of implementation to construct the second query which lets you win the round. If you need some help for the construction, it can be done in $O(N^2)$ by iteratively filling your paper row by row then column by column, while noting the last iteration that can affect the reply grid of a certain cell. Simply change the color of your paper at that iteration if you want to toggle that cell value of the reply grid to 1.

For $N = 3$, there is a strategy to win the round **expectedly** in 2 queries. On each query, simply send a randomly generated colored paper over all possible 2^9 possibilities. To calculate the probability of winning the round, note that the demon will see a random color from each hole in equal (half) probability. For each of these holes, the cell value of the result grid will be either flipped or stayed the same with equal probability as well. Therefore, the final cell value will be basically a coin flip, i.e. where you win if and only if it is head. If you want a more rigorous calculation to calculate the chance of losing the game after 999 queries, it is equivalent to flipping 999 coins and getting strictly less than 333 heads, which happens with probability less than 10^{-26}.

To sum up, you can win the full game with a total of 666 queries in expectancy, very cursed indeed.

D. Space Exploration

For this problem, we require the following procedures.

- Find an intersection point between two lines.
- Find intersections between a segment and a convex polygon. This can be done with the help of a binary search on the convex hull.
- Find the lines that cross a point that are tangent to a convex polygon. There should be two such lines if the point is outside the polygon or touching the polygon's vertex; or only one such line if the point is on the polygon's side. This can also be done with the help of a binary search on the convex hull, by considering whether the tangents are touching only the upper hull, only the lower hull, or both the upper and lower hulls.

The implementation of the above procedures will not be detailed in this analysis.

There are two major cases that we need to consider: travel with no direction change, and travel with one direction change. The former clearly requires less distance than the latter case.

For the case when no direction change is needed, then the segment from the start point and the end point must not intersect with the convex polygon (note that touching the polygon is fine). If there is no such intersection, then the answer is simply the Euclidean distance between the two points.

For the case when one direction change is needed, we first find the tangent lines of the convex polygon that are crossing the start point and the tangent lines of the polygon that are crossing the end point. Find a tangent line that crosses the start point intersects with a tangent line that crosses the end point, then the answer is the Euclidean distance from the start point to the intersection point then finally to the end point. There can be more than one such tangent lines intersection, in which you need only the one that minimise the Euclidean distances.

If none of the above cases holds (because both points are on opposing sides of the polygon with the same slope), then you can not reach the end point using only one direction change.

The total time complexity for this problem is $O((N+Q)\log(N))$.

E. Merge Not Sort

Observe what happens if we merge any two arrays A and B of length N consisting of a permutation of integers from 1 to $2N$.

- Let $A[P]$ be the first element in $A[1..N]$ which is greater than $A[1]$.
- Let $B[Q]$ be the first element in $B[1..N]$ which is greater than $B[1]$.
- If $A[1] < B[1]$, then the "block" $A[1..P-1]$ must be appended to C, as all of them must be less than $B[1]$.
- Else, the "block" $B[1..Q-1]$ will be appended to C instead.
- Repeat the procedure with the new A and B.

In the end, the C will consist of a series of appended blocks. We label these blocks to be $C = C_1 \cdot C_2 \cdot C_3 \cdot \cdots$. Observe that, no matter which the block of C_i comes from (either A or B), the first element of C_{i+1} must be the first element which is greater than the first element of C_i. Therefore, given the input C, our first step is to split it into blocks, which can be done in $O(N)$. To construct the A and B, we can take note of the length of each block and then assign each block to be either from

A or B such that the total length is exactly N for both. This can be done using a simple Knapsack DP in $O(N^2)$.

As an additional info, the original constraint for this problem is actually $N \leq 100,000$. Indeed, there exists an $O(N\sqrt{N})$ algorithm to solve the problem, and this is left to reader as an exercise.

The total time complexity for this problem is $O(N^2)$.

F. Maximize The Value

First, let's discuss how to solve this problem if all of the K are same.

We can construct an auxiliary array B with size M which contains operations that can affect A_K. If K is not between L_i and R_i, then the value of array B_i is 0. Otherwise, the value of array B_i is X_i. By using array B, a query can then be answered by returning "the maximum subarray sum in a given range" from this array. This query can be answered using a segment tree data structure.

Now, to solve the problem for any K, we can sort the queries based on K. Initially, we need to construct a segment tree for the auxiliary array when $K = 1$ and answer all queries with $K = 1$. When transitioning to next K, we can avoid rebuilding the segment tree by only updating the affected indices. If operation i has $R_i = K$, then we need to update B_i to 0. Another case is if operation i has $L_i = K + 1$, then we need to update B_i to X_i. Each operation i will only update the auxiliary array at most twice.

The total time complexity for this problem is $O((Q + M) \log M)$.

G. Grid Game 2

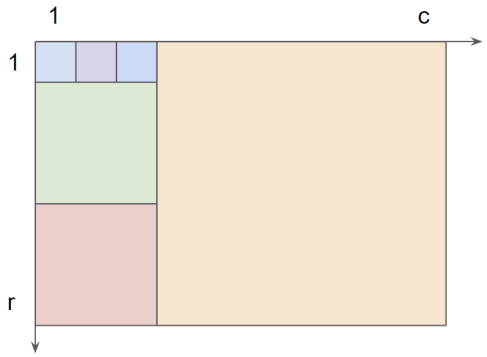
For convenience, we define $G_{r,c}$ be the game starting with a single black cell (r, c). Also define $S_{r,c}$ be the set of toggled cells (including (r, c)), if a player moves by choosing (r, c). Finally, define $S'_{r,c}$ as $S_{r,c} \setminus \{(r, c)\}$.

We will use the Sprague–Grundy theorem to solve this problem because the N black cells correspond to N independent games. To see why they are independent, consider a similar game where the only move in $G_{r,c}$ to be adding new games $G_{x,y}$, for all $(x, y) \in S_{r,c}$. This version of the game is clearly independent as we are only adding more games without affecting the state of other games. Next, suppose there is already a game $G_{x,y}$ and $(x, y) \in S'_{r,c}$ before the move. Recall that for a game G, the game sum $G + G$ is losing, hence after the move, $G_{x,y}$ can be ignored. Therefore, this version of the game is equivalent to the original toggling game.

A game $G_{r,c}$ has only a single move that can be performed on, and therefore the nimber is either 0

or 1. The key idea here is to not directly evaluate the nimber of a game, but rather the nimber sum of all the games from a *prefix*.

Observe that the nimber sum of all games $G_{x,y}$ for all $(x, y) \in S_{r,c}$ is always 1. This can be seen by noting that playing on $G_{r,c}$ will result into a losing state. Since the set $S_{r,c}$ can be visualised as a square covering a grid, we note from the following figure that each of the nimber sum of all the games in a square is exactly 1.



Therefore, we can count whether the number of squares covering the rectangle is odd or even. This can be done using an Euclidean-like algorithm.

Since we are able to calculate the nimber of the black cells spanning from $(1, 1)$ to (r, c), the nimber of a single black cell easily follows.

Following is the pseudocode to compute the nimber of a single black cell.

```
int euclid(int r, int c) {
  return c > 0 ? (r / c % 2) ^ euclid(c, r % c) : 0;
}
int nimber(int r, int c) {
  return euclid(r, c) ^ euclid(r-1, c) ^ euclid(r, c-1) ^ euclid(r-1, c-1);
}
```

The total time complexity for this problem is $O(N \log \max(R, C))$.

H. Twin Friends

Notice that the order of characters in A and B does not matter. What matters is the count of each letter in the alphabet in both strings. Let's represent each letter in the alphabet by an integer in the $[0, 26)$ range. Then, we define the following:

- $cnt_A[c]$ = the count of letter c in A, for each c in $[0, 26)$.

- $cntB[c]$ = the count of letter c in B, for each c in $[0, 26)$.

Let's solve the problem using dynamic programming. Let $DP(c, prevC)$ be the number of ways, if we only consider letters $[c, 26)$, and we already used $prevC$ letters c from B (matched with letter $c-1$ of A). We now want to "match" the letters c in A (there are $cnt_A[c]$ of them), with $cnt_A[c]$ characters from B. According to the rules, we can only use letters c or $c+1$ from B. Define the following counts:

- $curC$ = the count of letter c taken from B.
- $curC'$ = the count of letter $c+1$ taken from B, which is equal to $cnt_A[c] - curC$.

Then, the value is:

$$DP(c, prevC) = \sum F(curC, curC') \cdot DP(c+1, curC')$$

, where:

- $0 \le curC \le cnt_A[c]$,
- $prevC + curC \le cnt_B[c]$,
- $curC' \le cnt_B[c+1]$.

Here, $F(curC, curC')$ is defined as the number of ways to assign $cnt_A[c]$ letters c to A', with the corresponding $curC$ letters c and $curC'$ letters $c+1$ to B'. The value is the product of the following values:

- The number of ways to choose $cnt_A[c]$ positions for the letters c in A'. Because we only consider letters $[c, 26)$, there are $(cnt_A[c] + cnt_A[c+1] + ... + cnt_A[25])$ available positions. Let's simplify the expression using suffix sum $sum_A[c] = cnt_A[c] + cnt_A[c+1] + ... + cnt_A[25]$. So, the number of ways is $\binom{sum_A[c]}{cnt_A[c]}$.
- The number of ways to choose $curC'$ relative positions for the letters $c+1$ in B' matching with the corresponding positions of letters c in A', which is simply $\binom{cnt_A[c]}{curC'}$.

Therefore,

$$F(curC, curC') = \binom{sum_A[c]}{cnt_A[c]} \cdot \binom{cnt_A[c]}{curC'}.$$

By manipulating the conditions for valid pairs $(curC, curC')$, we can finally write out the equation for the DP as follows:

$$DP(c, prevC) = \sum \binom{sum_A[c]}{cnt_A[c]} \cdot \binom{cnt_A[c]}{curC'} \cdot DP(c+1, curC'),$$

for all $curC'$ between:

- $\max(0, cnt_A[c] - (cnt_B[c] - curC))$, and
- $\min(cnt_A[c], cnt_B[c+1])$, inclusive.

The base case is $DP(26, 0) = 1$, and the result is $DP(0, 0)$.

Computing this DP directly leads to a quadratic solution. To get the full points, we need a linear solution as follows. Observe that the equation can be rewritten as:

$$DP(c, curC) = \binom{sum_A[c]}{cnt_A[c]} \cdot \sum \binom{cnt_A[c]}{curC'} \cdot DP(c+1, curC')$$

The expression under the sigma symbol can be computed in constant time using an auxiliary table (hint: prefix/suffix sum of the DP). The rest is left as an exercise for the readers. Thus, the solution will run in $O(N)$ time.

The total time complexity for this problem is $O(N)$.

I. Contingency Plan 2

This problem asks for the minimum number of edges to be added in a polytree such that there is a unique topological ordering. As a reminder, a directed acyclic graph (DAG) has a unique topological ordering if and only if it has a Hamiltonian path.

Let **minimum chain decomposition** of the graph be the minimum number of disjoint paths such that each node from the graph belongs to one of the disjoint paths. The minimum number of edges that needs to be added is related to the size of minimum chain decomposition. Let $MCD = \{P_1, P_2, ..., P_{|MCD|}\}$ be the minimum chain decomposition of the DAG, where P_i represents the set of ordered nodes in path i of the decomposition. To get MCD, we can construct a new bipartite graph B and find the maximum matching in the B, detailed as follows.

1. For each i that satisfies $1 \leq i \leq N$ in the original graph, we create 2 new nodes in B, i.e. $outNode_i$ and $inNode_i$.
2. If there is an edge $u \to v$ in the original graph, then add an edge $outNode_u \to inNode_v$ in B.
3. Find a maximum bipartite matching of B.
4. If $outNode_u$ matches with $inNode_v$, then, in the original graph, node u and node v belong to in the same path. Moreover, v appears right after u in that path.

After we get MCD, there exists a correct order to chain the paths such that by adding an edge from the last node of P_i to the first node of P_{i+1} for each $1 \leq i < |MCD|$, will induce a Hamiltonian

path in the graph and no cycle is formed. To get the correct order to chain the paths, we need to construct a new graph G with $|MCD|$ vertices, where each node corresponds to a path in the MCD. For each edge from $u \to v$ in the original graph and $u \in P_i, v \in P_j$ such that $i \neq j$, we add an edge $i \to j$ in G. We claim that there is no cycle in G.

Proof. By contradiction, assume that there is a cycle of nodes, i.e. $s_1, s_2, ..., s_l, s_1$ in G. From the way we construct G, it implies that there is a directed edge from node $u \in P_{s_i}$ to $v \in P_{s_{i+1}}$ (for simplicity, let $s_{l+1} = s_1$) in the original polytree. Because s_i in G is a contraction of path P_i in the original polytree and there is a cycle in G, therefore by following the cycle in G, we will be able to find an undirected cycle in the polytree as well. However, a polytree has no cycle in its underlying undirected graph. Therefore, G does not contain a cycle. □

Because there is no cycle in G, we can find a topological ordering in G which corresponds to the paths ordering such that we can add an edge for every pair of endpoints of adjacent paths in the ordering. Thus, the minimum number of edges to be added is $|MCD| - 1$.

The complexity to get the find minimum chain decomposition is $O(N\sqrt{N})$ by using Hopcroft-Karp's (or Dinic's) algorithm. As to find the correct order, the complexity is $O(N)$.

The total time complexity for this problem is $O(N\sqrt{N})$.

J. Count BFS Graph

We will try to emulate the behaviour of BFS. We use the dynamic programming $DP(i, inQueue, isFirst)$ — how many ways to add edges into the graph if:

- the BFS has pushed $A[1..i-1]$ inside the queue;
- $inQueue$ number of $A[1..i-1]$ are still inside the queue;
- the front element of the queue has children (if $isFirst$ is $true$) or not (if $isFirst$ is $false$);
- we only consider adding edges to $A[1..i-1]$.

The DP transitions are as follows.

- Pop the queue and transition to $(i, inQueue - 1, true)$.
- Enqueue $A[i]$ by connecting it with the vertex that is in front of the queue. This can be done when the $isFirst$ is $true$ or $A[i-1] < A[i]$. Additionally, $A[i]$ can be connected with any of the other vertices in the queue. Hence, there are $2^{(inQueue-1)}$ ways multiplied by the transition to $(i+1, inQueue+1, false)$.

The base case are as follows.

- If $inQueue < 0$, there are 0 ways.
- Otherwise, if $idx > N$, there is 1 way.

The answer is then simply $DP(2, 1, true)$.

The total time complexity for this problem is $O(N^2)$.

K. Deck-Building Game

Let $XOR(S)$ be the xor of the deck whose card ids $i \in S$. We start with a simple yet crucial observation: Two disjoint decks S_1 and S_2 have the same XOR if and only if $XOR(S_1 \cup S_2) = 0$. Thus, this problem boils down to calculating the following formula: $\sum_S 2^{|S|} \cdot [XOR(S) = 0]$.

Denote $P(i)$ as a polynomial $(a_0 + a_i \cdot x^i)$, where a_j denote $\sum_S 2^{|S|} \cdot [XOR(S) = j]$ if we only consider cards whose value is i. Note that a_0 and a_i can be found via dynamic programming. Observe that the formula we want to calculate is the coefficient of x^0 in $P_1 \oplus P_2 \oplus \cdots \oplus P_{\max(A)}$, where $\oplus$ represents XOR convolution. We have to do $O(\max(A))$ XOR convolution, where the complexity of a single XOR convolution is $O(\max(A) \log(\max(A)))$ (i.e if we use FWHT) because the highest coefficient degree is $O(\max(A))$. Thus, this approach would run in $O((\max(A))^2 \log \max(A))$, which is too slow.

However, we can speed-up the whole process. Suppose we do the XOR convolutions in a Divide-and-Conquer fashion as shown below.

```
dnc(l, r):
    if l+1 = r:
        return P(l)
    m := (l + r) / 2
    lhs_polynomial := dnc(l, m)
    rhs_polynomial := dnc(m, r)
    return xor_convolute(lhs_polynomial, rhs_polynomial)
```

The result will be the coefficient of x^0 from $\mathrm{dnc}(0, \mathrm{twoCeil}(\max(A)))$, where $twoCeil(\max(A))$ denotes the smallest power of two that is greater than $\max(A)$. At a glance, this approach seems to be as bad as the previous approach. However, there are some important observations that can be made on $\mathrm{dnc}(0, \mathrm{twoCeil}(\max(A)))$. Proofs are omitted for brevity.

- At any point, $r - l$ must be a power of two

- The result of $\mathrm{dnc}(l, r)$ will be a polynomial whose non-zero coefficients must be on x^i where i falls in $[0, r-l)$ or $[l, r)$
- If we split the result of $\mathrm{dnc}(*, *)$ into two polynomials whose range are as described above, we can split the XOR convolution of $lhs\_polynomial$ and $rhs\_polynomial$ in $\mathrm{dnc}(l, r)$ into four XOR convolution. Furthermore, the result of each convolution will be a polynomial whose non-zero coefficients are on x^i where i falls in $[0, m-l)$, $[m-l, r-m)$, $[l, m)$, and $[m, r)$ terms. Moreover, the result of one convolution will be enclosed in exactly one of those segments.

Due to above observations, we can modify our XOR convolution such that the maximum degree of the polynomial we convolute to be $O(r-l)$. Thus, the complexity of the $\mathrm{dnc}(0, \mathrm{twoCeil}(\max(A)))$ will be $O(\max(A)\log^2(\max(A)))$.

The total time complexity for this problem is $O(\max(A)\log^2(\max(A)))$.

L. Palindromic Parentheses

Note that you can always choose all opening brackets (or closing brackets) as a palindromic subsequence of any balanced parentheses sequences. Furthermore, for a parentheses sequence to be balanced, its first character must be an opening bracket and its last character must be a closing bracket. Therefore, the answer does not exist if $K < \frac{N}{2}$ or $K = N$.

There are a lot of solution, and the following is one of the easier to implement.

If $K = \frac{N}{2}$ then the solution is `(((...()))...)`. Note that if we move one of the closing brackets in the middle to the front, i.e., `()(((...()))...)`, then the LPS will increase by 1. Similarly, if we move one of the opening brackets in the middle to the back, i.e., `(((...()))...)()`, then the LPS will increase by 1. We can alternately move the closing and opening brackets to increase the LPS one by one until $K = N - 1$, in which the solution is $()()() \ldots ()$.

The following are examples when $N = 10$.

- $K = 5$: `((((()))))`.
- $K = 6$: `()(((())))`.
- $K = 7$: `()((()))()`.
- $K = 8$: `()()(())()`.
- $K = 9$: `()()()()()`.

The total time complexity for this problem is $O(N)$.

M. Triangle Construction

Let $M = maxA_i$ and $S = \Sigma A_i$. Pick any side P where $A_P = M$. We then consider the following 2 scenarios.

In the first scenario, if $M > 2 \times (S - M)$, then the answer will be $S - M$. The proof is by construction as we can always create triangles with 2 points from side P and 1 point from the other sides. To make sure that the triangles do not intersect, we can iteratively pair the "rightmost" 2 points of side P with the "leftmost" point from the other sides, i.e. side $P + 1$. Note that this answer is already maximum as there will be degenerate triangles created from 3 points of side P afterwards.

In the second scenario, if $M \leq 2 \times (S - M)$, then we claim that the answer is $\lfloor \frac{S}{3} \rfloor$, i.e. it is always possible to construct the maximum possible triangles which are also non-degenerate and not intersecting. If $M = 1$, we can always create the triangles from 3 adjacent sides. If $M \geq 2$, we use the observation from the first scenario where we can create a triangle with the rightmost 2 points from side P and the leftmost 1 point from side $P + 1$. After that, form a new array A' consisting of unchosen special points and ignoring a side with zero special points. Note that with this new array A', S will be decreased by 3 and M could be decreased by up to 2. By assuming the new M is at least 2 (or otherwise we can start creating triangles from 3 adjacent sides) and the new S is at least 3 (or otherwise no more triangle can be made), array A' then will always fall again into the second scenario. (The only exception is $A = [3, 1, 3]$, however.) Repeat such construction to complete the proof.

Computing M and S can be done in $O(N)$, and calculating the final answer can be done with a simple "if" in $O(1)$.

The total time complexity for this problem is $O(N)$.