ICPC Jakarta 2021 - Problem Analysis

Problem Authors:

- Christopher Samuel
- Jonathan Irvin Gunawan
- Muhammad Ayaz Dzulfikar
- Prabowo Djonatan
- Rafael Herman Yosef
- Suhendry Effendy
- Winardi Kurniawan
- Wiwit Rifa'i

A. XOR Pairs

Let $f(x)$ be the number of pairs (A, B) such that $(A \oplus B) = x$ and $1 \leq A, B \leq x$. Observe that neither A nor B can be equal to x as the other then will be 0, thus, we can update the second constraint into $1 \leq A, B < x$. Then, the answer to this problem is simply $\sum_{i=1}^{N} f(i)\,[i \notin S]$.

The value of $f(x)$ is $2 \cdot (x - 2^{\lfloor \log_2 x \rfloor})$. How do we know? Well, generally, there are two ways to figure out how to compute function $f(x)$, i.e. deduce the formula by observing $f(x)$ for some small values of x, or by doing an analysis. For the sake of this editorial, we'll present an approach with the latter method.

We would like to count the number of pairs (A, B) such that $(A \oplus B) = x$ and $1 \leq A, B < x$. Without loss of generality, assume $A < B$, thus, $1 \leq A < B < x$. Later, we only need to double the output to consider the other symmetric case, $A > B$.

We show how to construct a valid B from x. Consider a binary representation of x and start from $B = x$. Flip one bit in B from 1 to 0, **except** for the leftmost bit of B (otherwise, then A will be larger than B). Let's call this flipped bit a pivot bit. This guarantees that B will be smaller than x and larger than A. Then, all bits to the right of the pivot bit can be assigned to any value (0 or 1) as the resulting B will never be larger than x. Therefore, the number of possible valid B with a pivot bit is equal to the value of that pivot bit.

As any bit 1 in x can be a pivot bit except for the leftmost one, then, the value of $f(x)$ is simply x without the leftmost bit, and multiplied by two (don't forget for the other symmetric case).

This solution runs in $O(N \log N + M)$ and can be optimized into $O(N + M)$. As a side note, there is a $O(M + \log N)$ solution for this problem, but it's not needed to get accepted in this problem.

B. Bicycle Tour

First, assign each edge to its value, i.e. the required power to traverse the edge, then construct the minimum spanning tree (MST) of the graph. Process all edges that are **not** part of the MST one by one in increasing order of their values. Let the edge being processed be (u, v). For each node on the path connecting u and v in the MST whose answer is still unknown, assign the node's answer to be the edge (u, v)'s value.

Unfortunately, this solution runs in $O(NM)$; not fast enough to pass the time limit.

There are several ways to speed up the solution. For example,

- Use a "node compression" technique, e.g., with a disjoint set data structure. All the nodes in the path between u and v are merged into a single node. So, the next time we visit any of these nodes, we can directly jump to their ancestor (since all the merged nodes already have their answer).
- Use the small-to-large technique.
- Use a data structure that supports the minimum path update in a tree.

The first solution runs in $O(M \log N)$ while the others run in $O(M \log^2 N)$. Any of the above solutions will get accepted.

C. Energy Generation

This problem can be simplified by first separating the 2-dimensional components and then treating them independently (each in a 1-dimensional problem).

To simplify the discussion, whenever we say "pair of towers", we mean a pair of towers that satisfies the requirement for an interesting phenomenon.

Let's represent a tower's orientation with a tuple $[a, b]$. A tower oriented at 0° has a positive a and positive b (i.e. 0° $\rightarrow [+, +]$), a tower oriented at 90° has a positive a and negative b (i.e. 90° $\rightarrow [+, -]$), 180°$\rightarrow [-, -]$, and 270°$\rightarrow [-, +]$. Now, let S be the Hamming distance of two tuples $[a_1, b_1]$ and $[a_2, b_2]$. For example, $[+, +]$ and $[+, -]$ has an $S = 1$ (only b component differs); $[-, +]$ and $[+, -]$ has an $S = 2$ (both a and b components differ).

Observe what happened to the interesting phenomenon when we use this representation: A pair of towers will generate an interaction energy of $2G(1 - S)$—note that it's for a pair of towers, each generates $G(1 - S)$ energy. Similarly, observe what happened to the cost of rotating a tower: A tower that is rotated from $[a_1, b_1]$ into $[a_2, b_2]$ will generate a passive energy of $P(1 - S)$.

To make our search easier, we can subtract $2G$ from each pair of towers (thus, making the interaction energy to be $-2GS$) and subtract P from each tower (thus, making the passive energy to be $-PS$). Then, the problem is reduced to finding the towers' configuration that minimizes $-2GS$ and $-PS$.

With this representation, the total generated energy only depends on how many components differ, thus, we can treat each component a and b independently.

The reduced problem for each component can be solved with a minimum-cut (maximum-flow) formulation.

1. Create a node for each tower, a source node s, and a sink node t.
2. If a tower is $+$, connect this node to the source node with a capacity of P; otherwise, connect to the sink node with a capacity of P.
3. For each pair of towers, connect them with a capacity of $2G$.
4. Find the minimum s-t cut of this graph.

The answer to this problem is $M \cdot 2G + N \cdot P - \text{mincut}(a) - \text{mincut}(b)$ where M is the number of pair of towers.

D. Uniform Maker

We should preserve the most frequent character in each position. Let f_i be the occurrence count of the most frequent character among characters at the i^{th} position. The answer for this problem is $\sum_{i=1}^{M}(N - f_i)$.

E. Concerto de Pandemic

Suppose we can find the minimum number of concerts needed to serve all the fans while each of them has a maximum traveling time of X, then we can perform a binary search on X to obtain the minimum needed traveling time when there are only at most P concerts. The former (sub)problem can be reduced into a **point interval covering problem**, i.e. given a set of intervals, find the minimum number of points such that each interval contains at least one point. Each interval represents a range of cities that can be reached within X traveling time by a fan, and each "selected point" represents a concert held in that city. An interval is covered by a point if and only if the point lies on the interval. Next, we'll discuss how to solve the subproblem.

To handle the cycle issue, simply append the list of fans (assumed to be sorted) with itself, i.e $[1..K] + [1..K] \rightarrow [1..2K]$. Then, we have to solve the point interval covering problem for each range of $[1, K]$, $[2, K+1]$, $[3, K+2]$, $\ldots$, $[K, 2K-1]$, and find the one that gives the minimum number of needed points/concerts. The greedy solution for the point interval covering problem runs in $O(K)$; it's not fast enough to pass the time limit as we need to do it for K times (i.e. for each range) causing the complexity to be $O(K^2)$.

One possible approach to speed up the solution is by using a **sparse table** data structure. Let $R[i][j]$ be an element of the sparse table indicating the nearest fan x ($> i$) that cannot be covered while all fans from i until $x-1$ are covered with exactly 2^j points and $x-1$ is maximum. The base case, $R[i][0]$, is the nearest fan x ($> i$) that cannot be covered together with fan i. $R[i][j]$ can be computed as follows: $R[i][j] = R[R[i][j-1]][j-1]$. With this data structure, the subproblem can be answered in $\Omega(K \log K)$. The exact complexity depends on how you implement the details, e.g., how to find the base case for the sparse table.

Some details are omitted from this editorial, but you should be able to figure those out.

F. Not One

First, let's handle the tricky case. If all nodes have a weight of 1, then the answer is 0; otherwise, the answer is at least 1. The remaining discussion focuses on the latter case.

There are several approaches for this problem and some of them rely on the observation that we only need to consider subtrees where the GCD of their nodes are primes. In this editorial, we will discuss an edge-based approach.

Assign each edge (u, v) to a value that is equal to $\mathrm{GCD}(A_u, A_v)$. Then, for each prime p, find the largest subtree (e.g., with a disjoint set data structure) by only considering edges that have p as its factor. Notice that any integer no larger than 10^6 has at most 7 unique primes, thus, each edge will only be considered in at most 7 different primes. If your method requires you to reset some values (e.g., the `parent[]` array in a disjoint set data structure), then be sure to reset only the affected nodes to avoid the time limit exceeded verdict.

G. Greedy Knapsack

First, let's address the trivial case: If the sum of all weights is no larger than the upper limit T, then the largest total value is simply equal to the sum of all values. The remaining discussion will be focusing on the more interesting cases where $\sum W_i > T$, and thus, not all items can be taken.

We will start with this lemma: The value for M that causes the greedy algorithm to return the largest total value must be in $[T - W_{max} + 1, T]$ where $W_{max} = \max\{W_i\}$. The proof is omitted.

Using this lemma, we can focus our search for the optimum M only on $[\max(1, T - W_{max} + 1), T]$. Note that repeatedly simulating the greedy algorithm even for this reduced range still will not pass the time limit as it has an $O(NW_{max})$ time complexity.

Our approach is by iterating through all items from 1 to N one by one while maintaining a set of valid ranges for M complete with their possible total value. At first, the only valid range is $[\max(1, T - W_{max} + 1), T]$ with a length of $H = \min(W_{max}, T)$.

When it's the i^{th} item's turns, we need to process all the ranges that can be used to take the i^{th} item. Let the range be $[L, R]$, then we only consider this range when $W_i \leq R$. There are 2 possible cases:

1. $W_i < L$. In this case, simply take the i^{th} item and update the range into $[L - W_i, R - W_i]$.
2. $L \leq W_i \leq R$. There are 2 cases that can happen to the greedy algorithm.
   * The item is taken $\rightarrow$ only happens when $M \in [W_i, R]$. Then, the range becomes $[0, R - W_i]$.
   * The item cannot be taken $\rightarrow$ only happens when $M \in [L, W_i - 1]$.

   Either way, we need to create both ranges to consider both situations.

Recall that we only consider a range $[L, R]$ when $W_i \leq R$ (as the i^{th} item cannot be taken otherwise). We can speed up the ranges-processing by employing a priority queue to store the ranges (larger R goes first), thus, only at most 1 range that cannot be used to take the i^{th} item that will be "visited" by the process. The number of operations then will be bounded by the total number of ranges created by the 2^{nd} case.

From the above operations, there will always be at most 1 range with $0 < L$, thus, there is only 1 range for the 1^{st} case. For the 2^{nd} case, observe that whenever we create new ranges, the total length of all ranges remains the same. As we start from a range with a length of H, then there can only be at most H valid ranges of length ≥ 1 in total (for all items). It means that the 2^{nd} case can only happen for $O(H)$ times.

This solution runs in $O(N \log H)$ where $H = \min(W_{max}, T)$. Some details are omitted from this editorial.

H. Cell Game

Aldo (the first player) can only win if and only if there is at least a pair of tokens of the same color having odd Manhattan distance to each other. Therefore, for Bondan (the second player) to not lose, he should rearrange the board such that each pair of tokens of the same color has even Manhattan distance; the best arrangement is a checkerboard.

One possible answer is by resizing the board into $2RC$ and putting tokens of the same color into same-color cells of a checkerboard (either white or black); this might not be the smallest board size but it serves as

the upper bound for our answer. We then need to check all possible sizes of the new board of $R' \times C'$ where $R' \geq R$, $C' \geq C$, and $R' \times C' \leq 2RC$ for a valid placement of all tokens that resulting in a non-losing configuration for the second player.

Let $f(x, b)$ be the minimum number of white cells needed to place tokens of the first x colors while there are b black cells available. Let A_x be the number of tokens with a color of x. Recall that we need to place all tokens of the same color on either white-only or black-only cells. The function $f(x, b)$ can be solved with the following recurrence relation (implemented with dynamic programming).

$$f(x,b) = \begin{cases} 0 & \text{if } x = 0 \\ f(x-1,b) + A_x & \text{if } b < A_x \\ \min(f(x-1,b) + A_x, f(x-1,b-A_x)) & \text{otherwise} \end{cases}$$

A board size $R' \times C'$ allows a valid non-losing configuration for the second player if and only if $f(26, B) \leq W$ where B and W are the number of black and white cells on the board, respectively. The tokens' placement for the output can be obtained by doing a backtrack from the function $f()$.

This solution runs in $O(mRC)$ where $m \leq 26$ is the number of different colors.

As the number of different colors in the game is quite small (only 26), instead of using a dynamic programming solution, an efficient implementation of a complete search solution is also fast enough to solve this problem.

I. Stable Planetary System

There are two possible cases for each pair of planets (i, j).

1. Both planets have a different revolution period. In this case, there must be a time $t \geq 0$ when both planets have a minimum distance that is equal to $|R_i - R_j|$, i.e. when they have the same angular coordinate to the star.
2. Both planets have the same revolution period. In this case, their distance will always stay the same (proof omitted, but try to imagine it).

To get the minimum distance for the 1^{st} case, we simply need to sort all the planets based on their radius and check the distance of any adjacent planets that have a different revolution period.

Finding the minimum distance for the 2^{nd} case is the classical **closest pair of points** problem that can be solved with a divide and conquer approach.

The answer we are looking for is the minimum among these two cases.

This solution runs in $O(N \log N)$. An $O(N \log^2 N)$ implementation is also accepted for this problem.

J. Feeder Robot

Let B be the last coop where the robot ends its movement (assume WLOG that $A \leq B$). All the coops from A to B must be filled with at least 1 pellet, thus, there are $M - (B - A + 1)$ pellets remain that must be somehow spent. Let L and R be the leftmost and rightmost coops that are visited by the robot, respectively;

then, $L \le A \le B \le R$. We want to distribute the remaining $M - (B - A + 1)$ pellets from coop L up to coop R such that the robot starts at A, ends at B, and there is a valid robot movement that corresponds to the distribution. Let $F(L, R, B)$ be the number of such possible distributions.

Let P_i be the amount of additional pellets that are distributed to coop i and coop $i + 1$; in other words, each count in P_i contributes to 2 pellets: 1 pellet at coop i, and 1 pellet at coop $i + 1$. The following constraints ensure that $P_{1..N-1}$ corresponds to a valid robot movement in respect to L, R, and B.

- $2 \cdot (P_1 + P_2 + .. + P_{N-1}) = M - (B - A + 1)$.
- $P_i = 0$, for $i < L$ or $R \le i$.
- $P_i \ge 0$, for all i where $A \le i < B$.
- $P_i \ge 1$, for all i where $L \le i < A$ or $B \le i < R$.

You might want to stop for a while and figure out why the last constraint is needed. Notice that $F(L, R, B)$ is equal to the number of possible valid assignments for $P_{1..N-1}$.

The number of valid assignments for $P_{1..N-1}$ can be computed with the following formula (derived with the stars and bars technique to handle the last constraint).

$$F(L, R, B) = \binom{(M - (B - A + 1))/2 + (B - A) - 1}{R - L - 1}$$

Observe that B should have a different parity than $A + M$; otherwise, the robot will never be able to ends at coop B. The equation can be rearranged into the following.

$$F(L, R, B) = \binom{(M + B - A - 3)/2}{R - L - 1}$$

For a given L and R, the possible range for B can be determined (actually, independent of L as B has to be at least A). Let G be the smallest and H be the largest possible $(B - A)$ for a given L and R.

$$F(L, R, *) = \sum_{\substack{G \le x \le H \\ (x-1) \equiv M \pmod 2}} \binom{(M + x - 3)/2}{R - L - 1}$$

As the bottom part of the binomial coefficient remains the same throughout the summation, then the series in this equation can be simplified with the hockey-stick identity.

$$F(L, R, *) = \binom{(M + H - 1)/2}{R - L} - \binom{(M + G - 3)/2}{R - L}$$

Now, instead of iterating through all possible pairs of L and R, we will iterate for all possible pairs of L and S where $S = R - L + 1$ (i.e. the number of elements between L and R). In other words, modify $F(L, R, *)$ into $F(L, S, *)$.

Observe that when we compute for L and $L + 1$ for the same S, the H value for $L + 1$ is either the same or $+2$ larger than the H value for L (this implies that the top part of the binomial coefficient will be at most

$+1$ larger for $L+1$). In particular, the changes in H exhibit an alternating pattern of length 2. On the other hand, G remains the same. We can utilize this property to compute $F(*, S, *)$ efficiently. To make sure the H value constantly increases in the series, we need to group the series in $F(*, S, *)$ into odd-indices and even-indices. For each group, we can exploit the hockey-stick identity again. Therefore, computing $F(*, S, *)$ can be done in $O(1)$.

Do it for each possible S, and apply the same idea for the other case ($B < A$).

This solution runs in $O(N + M)$.

K. White-Black Tree

Let the height of a node i, h_i, be the number of edges on the longest path from node i to one of its leaves. Also, let M_h be the number of white nodes in the tree that have a height of h. The first player loses if and only if M_h is even for every possible h.

Why? Observe that the end game (a losing state) has its M_h even for every possible h (to be exact, 0). If all values of M are even, then any valid move will make M_h for at least one h to be odd. If there exists a height k such that M_k is odd, then the respective player has a move that makes M_h become even for every possible h. In particular, the player needs to choose the highest white node with an odd M_h and flip that node. Then, the player needs to flip its predecessors as needed such that the remaining M_h becomes even.

This solution runs in $O(N)$.

L. Happy Travelling

Let $f(i)$ be the maximum total happiness that can be obtained by getting to city N if we start from city i. The following recurrence relation defines $f(i)$.

$$f(i) = H_i + \max_{i<j\le i+T_i} \left\{ f(j) - \left\lfloor \frac{j-i}{K} \right\rfloor \times D \right\}$$

Computing $f(i)$ directly as above requires an $O(N)$ time complexity causing the overall solution to be $O(N^2)$; not fast enough to get accepted. We need to speed up the computation.

Notice that the i and j components in the computation of $\lfloor (j-i)/K \rfloor$ can be split individually, and this opens up a possibility to speed up the computation for $f(i)$. However, there are two cases that we first need to consider.

(a) If $j < i \pmod K$, then $\lfloor (j-i)/K \rfloor = \lfloor j/K \rfloor - \lfloor i/K \rfloor - 1$.

(b) If $j \ge i \pmod K$, then $\lfloor (j-i)/K \rfloor = \lfloor j/K \rfloor - \lfloor i/K \rfloor$.

Here, we'll discuss one possible method to handle these cases. Specifically, we can use a 2-dimensional coordinate to store the data and employ a data structure to handle queries in 2-dimensional data.

Each coordinate $(j, j \bmod K)$ stores the value of $f(j) - \lfloor j/K \rfloor \cdot D$. Note that only these coordinates are used. To compute $f(i)$ we need to consider all coordinates $(j, j \bmod K)$ for $i < j \le i + T_i$. Although the

data only exists in the "diagonal", it will be much easier if we simply do a rectangle query and treat the non-existence elements as if they store $-\infty$. Therefore, to compute $f(i)$, we simply need to find the maximum among these two rectangles.

- The maximum among $(i+1, 0)$ to $(i+T_i, (i \bmod K)-1)$, and add $\lfloor i/K \rfloor \cdot D + D \rightarrow$ case (a).
- The maximum among $(i+1, i \bmod K)$ to $(i+T_i, K-1)$, and add $\lfloor i/K \rfloor \cdot D \rightarrow$ case (b).

With a proper data structure (e.g., 2D segment tree), the computation of $f(i)$ can be done in $O(\log^2 N)$. Thus, the total time complexity for this solution is $O(N \log^2 N)$.

M. Maxdifficent Group

The maximum possible largest $\mathrm{diff}(i, i+1)$ can only happen on these cases.

- The sum of a contiguous subarray that ends at i is maximum and the sum that starts at $i+1$ is minimum.
- The sum of a contiguous subarray that ends at i is minimum and the sum that starts at $i+1$ is maximum.

We can employ **Kadane's algorithm** to find the minimum/maximum sum of a contiguous subarray that ends at each i. To find the minimum/maximum sum of a contiguous array that starts at i, we can also employ Kadane's algorithm, but we need to process it in a reversed order (from N down to 1).

This solution runs in $O(N)$.