Ideafuse 2021 - Final Round

Problems Editorial

Problem Authors	Benny Wijaya
	Christoper Ganda
	Erick Kwantan
	Ricky Sentoso
	Suhendry Effendy
	Vincentius Madya
Editorial	Ricky Sentoso
	(edited by Suhendry Effendy)

A. Indivisible Bracket

A balanced bracket sequence starts with an open and ends with a close bracket.

Observe that an indivisible balanced bracket sequence must have its rightmost close bracket corresponding to the leftmost open bracket, otherwise, it is divisible. To find the longest indivisible balanced bracket sequence, we need to find the corresponding open bracket for each close bracket in S. This process can be sped up by the classic bracket matching algorithm using stack data structure as follows.

For each index i from 1 to $|S|$:

1. If S_i is an open bracket, push i to the stack.
2. If S_i is a close bracket and the stack is not empty, update $answer$ with $max(answer, i - stack.top() + 1)$ and pop the topmost element from the stack.

This solution runs in $O(|S|)$ per test case.

B. Power of 2 Sum

Let S be the sequence satisfying $2^{s_1} + 2^{s_2} + ... + 2^{s_{|S|}} = N$ where $s_i \in S$. We can build S' from S such that $|S'| = |S| + 1$ while satisfying the same constraint by replacing any s_i with two $(s_i - 1)$. Note that $2^{s_i} = 2^{(s_i-1)} + 2^{(s_i-1)}$.

First, build an initial sequence S_0 by considering the binary representation of N. Observe that S_0 contains the minimum number of elements needed to construct N. Next, let's look at these three separate cases:

- If $K < |S_0|$, then no sequence can satisfy the problem's constraints; the output is "No".
- If $K = |S_0|$, then S_0 is trivially the answer.
- If $K > |S_0|$, then we can extend the sequence repeatedly until its size equal to K. To ensure that a lexicographically smallest final sequence is produced, we need to greedily choose the largest element s_i to be split into two $(s_i - 1)$ in each step. It can be proven that under the given problem's constraints, a valid solution always exists; the proof is left as an exercise to the reader.

C. Connection Plan

Consider the following three cases while we compute the 1^{st} best and 2^{nd} best plan.

1. Only generator $\#1$ is used.
2. Only generator $\#2$ is used.
3. Both generators are used.

The first two cases are the classical second-best spanning tree problem, thus, we won't discuss it at length here.

For the third case, we need to find the minimum spanning forest (can be done with the Kruskal's algorithm) and stop when there are only two connected components left (actually, it is one component if we consider an imaginary edge connecting generator $\#1$ and generator $\#2$). Beware with a minimum spanning forest that does not use both generators; in such a situation, that minimum spanning forest shouldn't be considered as a candidate solution as it is already considered in the first two cases.

We can compute the second-best minimum spanning forest for the third case similar to how we solve the second-best spanning tree problem. However, do not consider any candidates that only use one generator.

Finally, the answer to this problem is the two best plans that are obtained from any of the three cases.

This solution runs in $O(NM)$ per test case.

D. Parity Game

We can combine a *minimax* algorithm with dynamic programming to solve this problem. The minimax algorithm finds the best move considering the other player's all choices, i.e. a player can only win if and only if there is a move such that the other player cannot win no matter what their (next) move is.

Let a function $f(L, R, T, P_1, P_2)$ be a boolean whether the player T can win the subgame where

- L and R denote the subgame range of $[L, R]$.
- $T \in \{\texttt{first}, \texttt{second}\}$ denotes the player's turn.
- $P_1 \in \{0, 1\}$ denotes the parity of all numbers sum collected by the first player.
- $P_2 \in \{0, 1\}$ denotes the parity of all numbers sum collected by the second player.

On each state, the player can take the prefix or suffix range of $[L, R]$. We simply need to sum and compute the parity for each prefix/suffix and update the P_1 or P_2 accordingly (depending on whose player's turn it is). The game ends when $L > R$ (no move can be made) and the return value is set according to T, P_1, and P_2.

However, notice that this dynamic programming solution has an $O(N^2)$ with a transition of $O(N)$ for each state—there are $O(N)$ prefix and suffix that need to be computed for each state. Therefore, the total time complexity for this solution is $O(N^3)$, not fast enough to pass the time limit for this problem.

In the previous solution, the player takes turns on each state (and we need to iterate through all prefix/suffix in the subgame range). We can relax the state by allowing the player to retain their move on the next state so we only need to consider at most 2 moves on each state, i.e. take the left-most or take the right-most element in the subgame range. To do this, we need an extra variable $S \in \{\texttt{LEFT}, \texttt{RIGHT}, \texttt{FREE}\}$ in the states to determine whether the player can only consider the left-most element, right-most element, or any of the two.

When a player performs a valid move on a state (constrained by S), they have two choices. The first one is to retain their turn; in this case, the next state should have S equals to whichever move they perform while T remains the same. The second one is to end their turn; in this case, the next state should have $S = \texttt{FREE}$ while T switches to the opponent.

With this additional dimension of a constant size (of 3) on the state, the transition is reduced from $O(N)$ to $O(1)$. Therefore, this solution runs in $O(N^2)$ per test case.

E. Numerophobia

This problem can be solved with dynamic programming by building the numbers satisfying the problem's constraints digit by digit.

For the sake of simplicity, let N_i represents the i^{th} most-significant digit of N, and $|N|$ represent the number of digits in N.

Let a function $f(i, L, F)$ be the number of unique non-zero elements not greater than N considering the digits from i to $|N|$. Variable $L \in \{\text{true}, \text{false}\}$ represents a flag indicating whether there exist an index $1 \leq j < i$ such that the j^{th} digit of the constructed number is strictly less than N_j. On the other hand, variable $F \in \{\text{true}, \text{false}\}$ represents a flag indicating whether the j^{th} digit of the constructed number is 0 for all $1 \leq j < i$; in other words, whether the first digit of the constructed number has NOT been placed.

$$f(i, L, F) = \begin{cases} 1 & \text{if } i > |N| \wedge F \text{ is false} \\ 0 & \text{if } i > |N| \wedge F \text{ is true} \\ g(i, L, F) & \text{if } i \leq |N| \wedge F \text{ is false} \\ f(i+1, \text{true}, \text{true}) + g(i, L, F) & \text{if } i \leq |N| \wedge F \text{ is true} \end{cases}$$

$$g(i, L, F) = \sum_{d=\text{lo}(F)}^{\text{hi}(i,L)} f(i+1, L \vee (d < N_i), \text{false}) \, [d \neq K]$$

$$\text{lo}(F) = \begin{cases} 0 & \text{if } F \text{ is false} \\ 1 & \text{if } F \text{ is true} \end{cases}$$

$$\text{hi}(i, L) = \begin{cases} N_i & \text{if } L \text{ is false} \\ 9 & \text{if } L \text{ is true} \end{cases}$$

Note that for the $(i \leq |N| \wedge F \text{ is true})$ case, we can choose not to place the first digit at the i^{th} index, hence, we add $f(i+1, \text{true}, \text{true})$ to the result.

The answer will be $f(0, \text{false}, \text{true})$. However, notice that the function $f(i, L, R)$ does not consider the case where the number 0 is one of the constructed digits. We need to adjust the answer (add at most 1) considering these cases.

- If $K = 0$, it is trivial that the number 0 will not appear.
- If $K \neq 0$, the number 0 appears only when $N \geq (10 \times K)$.

This solution runs in $O(|N|)$ per test case.

F. Best Product

Let's consider a solution using a segment tree data structure. Suppose we have found the maximum subarray product for two neighboring segments A and B; let segment AB be the

concatenation of segments A and B. How can we obtain the maximum subarray product for segment AB?

There are three cases that we need to consider.

- The maximum subarray product is in segment A.
- The maximum subarray product is in segment B.
- The maximum subarray product is composed of segment A's suffix and B's prefix.

For the last case, we only need to consider the following two candidates that result in a positive product.

- The min. negative product of A's suffix and the min. negative product of B's prefix.
- The max. positive product of A's suffix and the max. positive product of B's prefix.

These values can be computed linearly for each segment in the segment tree; the total time complexity to compute these values is $O(N \log N)$. Alternatively, you can speed up this computation into $O(N)$ by considering only a few cases (can you figure this out?).

Observe that the only case where the output is a negative integer is when there is only 1 element in the segment and its value is negative; this case can be handled separately.

Since we know how to merge two neighboring segments, we can construct a segment tree storing the maximum subarray product of any arbitrary range. Each query can be answered in $O(\log N)$. This solution runs in $O(N \log N + Q \log N)$.

There also exists a greedy solution that runs in $O(N + Q)$ for this problem that we didn't discuss here.

G. Bounded Jump

This problem is effectively about handling queries in a 2-dimensional data structure with $i = 1..N$ in the x-axis and S_i in the y-axis. However, we will discuss an approach that doesn't use any 2-dimensional data structure in this editorial.

Let a function $f(i)$ where $i = 1..N$ be the length of the longest L-K-bounded jump where i is the first element in the L-K-bounded jump of S. The following recurrence relation defines $f(i)$ formally.

$$f(i) = \left(\max_{i<j\leq \min(i+L,N)} f(j) \left[|S_i - S_j| \leq K \right] \right) + 1$$

Solving the recurrence relation with naïve dynamic programming requires $O(NL)$ running time per test case; not fast enough to get accepted. Fortunately, there is a way to improve the solution.

Consider a bottom-up implementation of the dynamic programming. Compute $f(i)$ one by one for each i from N down to 1. To compute $f(i)$, we need to find the largest $f(j)$ where $j = i + 1 \dots \min(i + L, N)$ while satisfying $|S_i - S_j| \leq K$. We will handle this with a data structure supporting a range maximum query on S_x, e.g., segment tree.

The segment tree contains only the largest $f(\cdot)$ that corresponds to each S_x. There are two issues. First, the constraint for S_x is too large (10^9), thus, we need to compress it; there are only $50\,000$ different values for S_x. Second, observe that there might be multiple indices j that have the same S_x while only the largest one is stored in the segment tree. We need to be able to update the largest $f(\cdot)$ for each S_x to satisfies the range for $j = i+1 \dots \min(i+L, N)$, e.g., the largest $f(\cdot)$ for an S_x might be $f(m)$ where m is outside the range, so $f(m)$ should be replaced with another $f(\cdot)$ that satisfies the range and also corresponds to S_x. This can be done if we maintain a list for each S_x that contains all $f(\cdot)$ that correspond to it and update them with a sliding window technique. Updating these lists can be done efficiently if we implement them with BBST, e.g., C++ `std::multiset` or `std::priority_queue` with lazy pop. The segment tree also needed to be updated to reflect these lists.

Now, right before we compute $f(i)$, the segment tree contains only the largest $f(j)$ for each S_x while j satisfies $i < j \leq \min(i + L, N)$. Then, to compute $f(i)$, we simply do a range maximum query from $S_i - K$ to $S_i + K$ on the segment tree.

This solution runs in $O(N \log N)$ per test case. It is fast enough to pass the time limit for this problem.

H. Cycle Home

We can model the problem with a state graph where each node is defined by three parameters $\langle r, c, b \rangle$ where:

- r denotes the row where Christopher is located.
- c denotes the column where Christopher is located.
- b denotes the current battery power.

The edges can be constructed following the problem description. In this problem, each node has a small number of edges (at most 8, on each direction and with each cycling mode). Note that if an edge goes to a non-existence node (e.g., $b < 0$), then such an edge is not valid.

There are $O(N^2M)$ nodes and $O(N^2M)$ edges in such a graph. The initial state is $\langle r_s, c_s, B \rangle$ where (r_s, c_s) is Christoper's initial location. The goal state is any of $\langle r_t, c_t, * \rangle$ where (r_t, c_t) is Christoper's home location.

Let's define the shortest path to a node in the graph as the minimum stamina unit that is required to reach the node from the initial state. The shortest path from the initial state to any other

states in the graph can be computed with the Dijkstra's algorithm or the 0 - 1 BFS algorithm since the weight is only 0-motor or 1-pedal.

If there is a state $\langle r_t, c_t, * \rangle$ for any valid battery power that has a shortest path of no greater than S, then print "YES"; otherwise, print "NO".

This solution runs in $O(N^2M)$ (with 0 - 1 BFS) per test case.

I. Screening Test

Observe that the XOR of two non-negative integers is odd if their rightmost binary digits are different. Let t be the last digit of B. To get the answer for this problem, we simply need to check for each i from $|B|$ to $|A|$ and increment the answer if $A_i \neq t$.

J. Crane Delivery

Consider each crane as a node in a graph where there is an edge between two nodes/cranes if and only if the sum of their operating radii does not exceed their distance. Then, this problem is equivalent to finding the minimum number of nodes to be traversed in the classical shortest path problem. Note that we cannot use a node if its weight limit is lower than the good's weight in the query.

Unfortunately, running BFS to compute the shortest path for each query will not pass the time limit.

Two observations must be made. First, there are at most only five unique crane's weight limits, i.e. $w_{1..5}$. We can generate five different graphs where only nodes that can carry at least w_i weight are considered in the respective graph. Second, there is only one destination (the warehouse), thus, we can precompute the shortest path cost from the warehouse node to any other nodes in one run for each graph.

To answer each query, we first need to find the smallest weight limit not less than the good's weight to determine which graph should be used. Then, we iterate through all possible starting nodes in the corresponding graph and find the one with the minimum cost. A node is a possible starting node if and only if the respective crane can reach the good and its weight limit is no less than the good's weight.

Beware of a tricky case where we can reach the warehouse using a crane that is not the warehouse's one. This can be handled easily if we create a new goal node that does not represent any of the cranes (i.e. separating the warehouse's crane and the warehouse into two different nodes), thus, multiple cranes might have an edge to this goal node.

This solution runs in $O(N^2 + QN)$.

K. Flower Field

A naïve solution is to check every possible subrectangle. However, this solution runs in $O(N^3M^3)$ per test case which is too slow for this problem. We can further optimize this solution with the *cumulative sum matrix* technique to bring down the time complexity into $O(N^2M^2)$ per test case, but it's still not enough to pass the time limit for this problem.

Let's first consider an easier version of this problem, i.e. a 1-dimensional problem (or $N = 1$). For the sake of simplicity, we refer to a subrectangle as a subarray. Observe that we don't need to check all the $\binom{M}{2}$ subarrays. For each subarray with ending index r from 1 to M, we only need to consider the largest subarray whose sum is not greater than K. This can be done in $O(M)$ with the sliding window technique.

To extend this solution for the original problem, we simply need to check all $\binom{N}{2}$ pairs of starting and ending row indices and treat each subproblem as a 1-dimensional problem. For each pair (i, j) of starting and ending row indices, the sum of $A_{(i..j),c}$ for a column c can be pre-computed with the *partial sum* technique. This solution runs in $O(N^2M)$ per test case and is fast enough to get accepted.

L. Even Balls

We can solve this problem by considering these two cases:

- If the sum of all integers in the input is even, then it is the answer.
- If the sum of all integers in the input is odd, then to get the largest possible even score, we need to subtract it with the smallest odd integer in the input. There must be at least one odd integer in the input, otherwise, the sum cannot be odd.