IOI 2022 Problems Discussion

Gunawan, Jonathan Irvin
Jump Trading Pacific Pte. Ltd.
jonathanirvingunawan@gmail.com

Djonatan, Prabowo
Garena Online Pte. Ltd.
prabowo1048576@gmail.com

Aji, Alham Fikri
IA TOKI
afaji321@gmail.com

Dzulfikar, Muhammad Ayaz
National University of Singapore
muhammad.ayaz97@gmail.com

Lie, Maximilianus Maria Kolbe
BINUS University
maxhaibara97@gmail.com

Mushthofa
IPB University
mush@apps.ipb.ac.id

Nurrokhman, Abdul Malik
Gadjah Mada University
abdul1024malik@gmail.com

Rifa'i, Wiwit
Google Taiwan Engineering Limited
wiwitrifai@gmail.com

Yudhiono, Hocky
IA TOKI
hocky.yudhiono@gmail.com

August 13, 2022

Below are the official solutions used by IOI 2022 Scientific Committee. Note that there might be more than one solution to some subtasks. Also, note that the order of the subtasks in the discussion **might not be ordered** for ease of discussion.

Since the purpose of this editorial is mainly to give the general idea to solve each subtask, we left several (implementation) details in the discussion for the reader's exercise.

1 Catfish Farm

Written by: **Lim Rui Yuan** (Singapore), NUS High School
Prepared by: Mushthofa, Prabowo Djonatan
Solutions, review, and other problem preparations by: Jonathan Irvin Gunawan, Muhammad Ayaz Dzulfikar, Wiwit Rifa'i
Analysis author: Muhammad Ayaz Dzulfikar

Recall that a pier of length k covers cells from row 0 to row $(k-1)$.

1.1 Subtask 1

In this subtask, all catfish are located in even-indexed columns.

We can build piers of length N in all odd-indexed columns to catch all the catfish. Thus, the answer for this subtask is the total weight of all catfish.

Time complexity: $O(M)$

1.2 Subtask 2

In this subtask, all catfish are located at either column 0 or column 1.

When $N = 2$, the optimal solution is to either build a pier of length N in column 0, or a pier of length N in column 1. This way, we either get all the catfish in column 1, or all the catfish in column 0.

When $N > 2$, apart from above solution, another possible optimal solution is by building a pier of length k ($k < N$) in column 1 and a pier of length N in column 2. Thus, we can catch the catfish of some prefix in column 0 and catfish of some suffix in column 1.

Therefore, the solution for this subtask is the maximum among all those possible optimal solutions.

Time complexity: $O(N + M)$

1.3 Subtask 3

In this subtask, all catfish are located at row 0.

For each column, we either do not build a pier, or build a pier of length 1. Notice that whether a catfish in column i is caught or not only depends on our decision in column $i-1$, i, and $i+1$. Thus, we can formulate a dynamic programming (DP) with the following states: the index of the current column and the decision in the last two columns.

Time complexity: $O(N + M)$

1.4 Subtask 4

In this subtask, $N \leq 300$ and $Y[i] \leq 8$ for all catfish.

The idea is an extension of subtask 3. Denote Y_{max} as the maximum value from array Y. Notice that there exists an optimal solution in which the piers' length is at most $Y_{max} + 1$. Thus, we can change our DP formulation to instead track the index of the current column and the piers' length in the last two columns. To speed up the calculation of the total catfish weight, we can build a prefix sum for them.

Time complexity: $O(N \times {Y_{max}}^3 + M)$.

1.5 Subtask 7

In this subtask, there are at most 2 catfish in each column.

The idea is to extend the solution from subtask 4 using the following observation:

Observation 1.1. *Let $R(i)$ be the set of rows containing catfish in column i. Then, there exists an optimal solution where for each $0 \leq i < N$ we either do not build a pier in column i, or we build a pier in column i of length r, where $(r - 1) \in (R(i - 1) \cup R(i + 1))$.*

The intuition is that it is better to have the end of the pier to be directly connected to the left or right of a catfish. If it is not, then we can just shorten it until it is.

Thus, for each column, there are at most 4 rows that we must consider to be the end of the pier. Then, we can plug in the solution from subtask 4 to solve this subtask.

Time complexity: $O(N + M)$

1.6 Subtask 5

In this subtask, we have $N \leq 300$.

The crux of the previous solutions is that we need to keep track of the piers' length from the previous two columns. To tackle this, first, we start with some definitions.

Definition 1.1. *A valley is an index i, such that:*

- *We build piers in column $(i - 1)$, i, and $(i + 1)$, and*
- *The length of the pier in column i is shorter than the pier in column $(i - 1)$ and column $(i + 1)$.*

Definition 1.2. *A bitonic sequence of piers is a continuous segment of piers in which there is no valley in it.*

Finally, we use the following observation:

Observation 1.2. *There exists an optimal solution with no valley in it. In other words, there exists an optimal solution where the constructed piers form several disjoint bitonic sequences.*

Proof. Observe that valleys will not contribute at all to the total weight; Thus, it is better to not build a pier in column i, as there might be catfish that can be caught if we do not build that pier at all. □

Thus, we can reformulate our DP states to be the index of the column, the length of the previous columns' pier, and whether right now the length is increasing or decreasing. As one of the dimensions changes from $O(N)$ to $O(1)$, the time complexity becomes $O(N^3 + M)$.

Time complexity: $O(N^3 + M)$

1.7 Subtask 6

In this subtask, we have $N \leq 3000$.

It is possible to speed up the DP from Subtask 5. Rather than trying all possible lengths for the current column, we can find the optimal length by storing the prefix maximum and suffix maximum of the DP from the next column. This will remove another $O(N)$, hence the time complexity becomes $O(N^2 + M)$.

Time complexity: $O(N^2 + M)$

1.8 Subtask 8

To fully solve this problem, we need to combine Observation 1.1 and Observation 1.2. Observation 1.1 implies there are only $O(N + M)$ possible DP states. Meanwhile, Observation 1.2 (and optimization from Subtask 6) implies all of the states can be computed in $O(1)$. This yields a solution with time complexity $O(N + M \log M)$ (due to sorting), which is sufficient to solve this problem.

Time complexity: $O(N + M \log M)$

2 Prisoner Challenge

Written by: **Hirotaka Yoneda** & **Masataka Yoneda** (Japan), The University of Tokyo
Prepared by: Wiwit Rifa'i
Solutions, review, and other problem preparations by: Abdul Malik Nurrokhman, Jonathan Irvin Gunawan, Prabowo Djonatan
Analysis author: Wiwit Rifa'i

This problem is about generating a finite state machine to compare two different integers, A and B, with the minimum number of states.

2.1 Subtask 1

In this subtask, we can choose $x = N$. We can use the following strategy:

- The first prisoner will always read 0 on the whiteboard, and then overwrites it with the number of coins in bag A.
- The second prisoner will read a positive integer which represents the number of coins in bag A. So, the second prisoner has to check bag B and then answer which bag has fewer coins.

2.2 Subtask 2

Let $S = \lceil \sqrt{N} \rceil$. In this subtask, we can do the similar thing in subtask 1 but with 2 phases instead of 1 phase. First, we can compare $\lfloor \frac{A}{S} \rfloor$ and $\lfloor \frac{B}{S} \rfloor$. If the values are the same, then we can continue to compare $(A \bmod S)$ and $(B \bmod S)$. This solution only needs $x = 2 \times S$.

There's also another solution that needs $x = 3 \times S$ using the fact that every non-negative number can be represented uniquely as $p^2 + q$ for non-negative numbers p, q such that $q < 2 \times p + 1$. First, we can compare $\lfloor \sqrt{A} \rfloor$ and $\lfloor \sqrt{B} \rfloor$. Then, if the values are the same, we can compare $A - \lfloor \sqrt{A} \rfloor^2$ and $B - \lfloor \sqrt{B} \rfloor^2$.

2.3 Subtask 3

In this subtask, we can get some partial scores depending on how small the chosen x is. The following are some possible solutions that could get some points from this subtask.

2.3.1 Solution with $x = 3 \times \lceil \log_2 N \rceil - 1 = 38$

We can simulate how to compare two binary numbers. By noting that it requires up to 13 bits to represent a number up to 5000, the value on the whiteboard represents the current state, i.e.

- If the value is $3 \times d$, then we need to check the $(12 - d)$-th bit of A, and then overwrite it with $3 \times d + 1$ or $3 \times d + 2$ depending on the bit value.
- If the value is $3 \times d + 1$ or $3 \times d + 2$, then we know the $(12 - d)$-th bit of A is either 0 or 1 from the previous prisoner. Then, the current prisoner has to check the $(12 - d)$-th of B and compare the

current bit value between A and B. If the values are still the same, the prisoner can overwrite it with $3 \times (d+1)$ to continue to the next bit. Otherwise, the prisoner can answers which bag has fewer coins.

Therefore, this solution needs $x = 3 \times \lceil \log_2 N \rceil - 1$ or $x = 38$ for $N = 5000$.

2.3.2 Solution with $x = 2 \times \lceil \log_2 N \rceil = 26$

Instead of storing the bit only from A on the whiteboard, we can store the bit of either A or B alternately. We can use the parity of the current bit position i to determine whether we are storing the i-th bit of A or B. Therefore, we don't need an additional state for when we have not stored any value of i-th bit.

2.3.3 Solution with $x = 3 \times \lceil \log_3 N \rceil = 24$

We can compare them using base 3 instead of base 2 using a similar idea as the previous solution.

2.3.4 Solution with $x = 3 \times \lceil \log_3 N \rceil - 2 = 22$

We can prune two states when processing the last digit (in base 3 representation). If the last digit is 0 or 2, then we can be sure whether $A < B$ or $A > B$ without needing to check the last digit from the other bag since we know that $A \neq B$.

2.3.5 Solution with $x = 3 \times \lceil \log_3 (N+1) \rceil - 3 = 21$

In the previous solution, we use the pruning idea on the last digit only. If we use the ternary number system like the previous solution, we can visualize it as a perfect ternary tree where we divide range $[0, 3^k - 1]$ into 3 parts recursively. And, the pruning in the previous solution is only applied for some leaf nodes (last digits).

Instead of pruning only the last digit, we can apply it to any possible range. Let's say we know that both A and B are currently inside the range $[L, R]$, and we are checking bag A. If we know that $A = L$, then we immediately know that $A < B$. If we know that $A = R$, then we know $A > B$. It means that we can "prune away" the leftmost and the rightmost integer from the range. After that, we can divide the range $[L + 1, R - 1]$ into 3 parts and determine which sub-range A belongs to. Then, the next prisoner can check bag B and determine which sub-range it belongs to. If both bags still belong to the same sub-range, then we can continue the process with this smaller range. Otherwise, we can already determine which bag has fewer coins. We continue this process until we can determine the answer.

When we divide a range into 3 sub-ranges, we need 3 additional states. If the base range is a range with 2 integers, then $3 \times K$ states can cover $3 \times (3 \times (\ldots 3 \times (2) + 2 \ldots) + 2) + 2 = 3^{K+1} - 1$ numbers. So, we need approximately $3 \times K = 3 \times (\lceil \log_3 (N+1) \rceil - 1) = 21$ states for $N = 5000$.

2.3.6 Solution with $x = 20$

We can optimize the previous solution. Dividing a range by 3 is not always optimal. We can use dynamic programming to determine what is the best divider for each level such that it can cover the largest range.

$$\texttt{dp}[x] = \max_{1 \leq i \leq x} (i \times \texttt{dp}[x - i] + 2)$$

where $\texttt{dp}[x]$ denotes the maximum range that can be covered using x states and $\texttt{dp}[0] = 2$. For $x = 20$, we can cover until $N = 5588$. Such x is the minimum such that $N \geq 5000$. It can be achieved by dividing the range $[1, 5588]$ by 3, 3, 3, 3, 3, 2, 2, and 1 for each level recursively using the same process as the previous solution.

3 Radio Towers

Written by: **Kevin Luiz Ponte Pucci** (Portugal), Oporto University
Prepared by: Jonathan Irvin Gunawan
Solutions, review, and other problem preparations by: Abdul Malik Nurrokhman, Muhammad Ayaz Dzulfikar, Prabowo Djonatan
Analysis author: Jonathan Irvin Gunawan

3.1 Subtask 1

In this subtask, the height of the towers is a bitonic sequence.

Let k be the index such that $H[k-1] < H[k] > H[k+1]$. It is impossible to lease two towers, both with indices not more than k. It is also impossible to lease two towers, both with indices not less than k.

If $R \leq k$ or $L \geq k$, then there is no way to lease more than one tower. Otherwise, we check whether we can lease both tower L and tower R using tower k as an intermediary.

Time complexity: $O(N + Q)$

3.2 Subtask 2

In this subtask, $Q = 1, N \leq 2000$.

Observation 3.1. *We can greedily try to lease towers from the lowest height.*

Proof. We can prove this by contradiction. Suppose tower x is the tower of the lowest height that the greedy solution can lease, but there is a more optimal solution that does not lease building x. Let a and b be the nearest tower to the left and right of tower x that is higher than tower x and is leased in the optimal solution. If the intermediary tower for towers a and b is to the left of tower x, then we can lease tower x and not lease tower b instead. Similarly, if the intermediary tower for towers a and b is to the right of tower x, then we can lease tower x and not lease tower a instead. This means that the greedy solution will not produce a less optimal solution. □

Definition 3.1. *For each tower x, $L[x]$ (and $R[x]$) is the nearest tower to the left (and to the right) of tower x with the height of at least $H[x] + \delta$.*

We iterate towers from the lowest height, and we can lease tower x if and only if there is no previously leased tower between towers $L[x]$ and $R[x]$. A naive implementation of this solution runs in $O(QN^2)$.

Time complexity: $O(QN^2)$

3.3 Subtask 3

In this subtask, $Q = 1$.

To solve this subtask, for each x, we can find $L[x]$ and $R[x]$ in $O(\log N)$ using segment tree. We can also get the minimum tower height between towers $L[x]$ and towers $R[x]$ using segment tree, and check whether there is a tower with a lower height than tower x.

Time complexity: $O(QN \log N)$.

3.4 Subtask 4

In this subtask, $D = 1$.

Doing the greedy solution will lease towers whose both of the neighbouring towers are higher than them. We can precompute which towers have higher neighbouring towers and use prefix sum to answer the queries.

Time complexity: $O(N + Q)$

3.5 Subtask 5

In this subtask, $L = 0, R = N - 1$.

Definition 3.2. *Let $LS[x]$ and $RS[x]$ be the nearest tower to the left (and to the right) of tower x with a lower height than tower x.*

In order for tower x to be leased, there must be a tower between towers $LS[x]$ and x (also between towers x and $RS[x]$) with a height of at least $H[x] + \delta$. Therefore, the value of δ must be at most $min(h_1, h_2) - H[x]$, where h_1 is the maximum height of towers between towers $LS[x]$ and x, and h_2 is the maximum height of towers between towers x and $RS[x]$.

To answer the queries, we can use binary search to count how many towers x such that tower x is leased when the value of δ is at most D.

Time complexity: $O((N + Q) \log N)$

3.6 Subtask 6

In this subtask, the value of D is constant among all queries.

Let us solve the task for a fixed value of D. Let $A_0, A_1, \ldots$ be the index of the towers leased in the greedy solution for $L = 0, R = N - 1$ question.

For $L = l, R = r$ question, let i, j be the values such that $A_{i-1} < l \leq A_i \leq A_{i+1} \leq \ldots \leq A_j \leq r < A_{j+1}$. The towers leased in the greedy solution for this question are towers $A_i, A_{i+1}, \ldots, A_j$ and possibly two other towers: one between towers l and A_i, and another one between towers A_j and r.

Let us focus on whether we can find a tower x between towers l and A_i that we can lease. We need tower x to be able to communicate with tower A_i, so there must be an intermediary tower between them. This means $R[x]$ must be less than A_i and $L[A_i]$ must be more than x. To check whether there exists such x, we check whether $(\min_{l \leq k \leq L[A_i]-1} R[k]) < A_i$. We can get the left-hand value using segment tree.

Finding a tower between towers A_j and r can be done similarly.

Time complexity: $O((N + Q) \log N)$

3.7 Subtask 7

Let us solve this task for different possible values of D.

For a given value of D, we can reuse the solution for subtask 5 to get the index of leased towers $A_0, A_1, \ldots$ for $L = 0, R = N - 1$ question. Similar to subtask 6, for $L = l, R = r$ question, let i, j be the values such

that $A_{i-1} < l \leq A_i \leq A_{i+1} \leq \ldots \leq A_j \leq r < A_{j+1}$. The values of A_i, A_j, and $j - i$ can be computed using a 2D data structure or persistent segment tree.

To find tower x (an additional tower between towers l and A_i that we can lease), we want to check whether there exists a tower k ($x < k < A_i$) such that $H[k] > \max(H[x], H[A_i]) + D$. We can do this by having another segment tree that computes: in a node that covers the range $[l', r']$, the segment tree computes $\max_{l' \leq i \leq j \leq r'} H[j] - H[i]$. If the value of the node that covers $[l, L[A_i]]$ is at least D, then there exists x, y where $l \leq x \leq y \leq L[A_i]$ and $H[y] - H[x] \geq D$. Therefore, we can lease tower x and tower y can be used as an intermediary between tower x and tower A_i.

Finding a tower between towers A_j and r can be done similarly.

Time complexity: $O((N + Q) \log N)$.

4 Digital Circuit

Written by: **Prabowo Djonatan** (Indonesia), Garena Singapore
Prepared by: Prabowo Djonatan
Solutions, review, and other problem preparations by: Jonathan Irvin Gunawan, Muhammad Ayaz Dzulfikar
Analysis author: Prabowo Djonatan

For ease of discussion, we will represent the circuit as a tree, where the gates are the nodes. Furthermore, the tree is rooted at 0, and the array P represents the parent of each node. We will also refer threshold gates as internal nodes, and source gates as leaves.

4.1 Subtask 1

In this subtask, $N = 1$, $M \leq 1000$, $Q \leq 5$.

The tree is star-shaped. For a node with c children, if the state of exactly k children is 1, then there are exactly k threshold parameters that can be assigned such that the node has state 1, which are $1, 2, \ldots k$. Since the root has all the leaves as its children, it is sufficient to output the sum of all A after each update.

Time complexity: $O(QM)$

4.2 Subtask 2

In this subtask, $N, M \leq 1000$, $Q \leq 5$, and the tree is a full binary tree.

Define $\mathtt{dp}_0(u)$ and $\mathtt{dp}_1(u)$ as the number of ways to make node u has value 0 and 1 respectively, by assigning parameters to the subtree of u.

We have the following transitions:

- $\mathtt{dp}_0(u) = \mathtt{dp}_0(l)\mathtt{dp}_1(r) + \mathtt{dp}_1(l)\mathtt{dp}_0(r) + 2\mathtt{dp}_0(l)\mathtt{dp}_0(r)$
- $\mathtt{dp}_1(u) = \mathtt{dp}_0(l)\mathtt{dp}_1(r) + \mathtt{dp}_1(l)\mathtt{dp}_0(r) + 2\mathtt{dp}_1(l)\mathtt{dp}_1(r)$

where l and r are the children of u.

After each update, recompute the whole `dp`.

Time complexity: $O(Q(N + M))$

4.3 Subtask 3

In this subtask, $N, M \leq 1000$, $Q \leq 5$.

Use the same definition of `dp` from subtask 2, and modify the transition to a knapsack-like DP to find the number of ways such that exactly k of its children have state 1. Alternatively, we can also use Left-Child Right-Sibling tree DP technique.

Time complexity: $O(Q(N + M)^2)$

4.4 Subtask 4

In this subtask, the tree is a perfect binary tree, and each update toggles only a single leaf.

A perfect binary tree has a height of $O(\log(N+M))$. Using the same DP formulation from subtask 2, we notice that the only DP values that are to be updated are those in the path from that leaf to the root.

Time complexity: $O(N+M+Q\log(N+M))$

4.5 Subtask 5

In this subtask, the tree is a perfect binary tree.

Suppose all leaves contained in the subtree of node u are updated. In that case, we can swap the value of $\texttt{dp}_0(u)$ and $\texttt{dp}_1(u)$, then lazily update its subtree. Thus, we can treat the whole perfect binary tree as a segment tree, and do the updates accordingly.

Time complexity: $O(N+M+Q\log(N+M))$

4.6 Subtask 6

In this subtask, the tree is a full binary tree (each node has exactly two children).

For each node v, note that the value of $\texttt{dp}_0(v)+\texttt{dp}_1(v)$ is fixed regardless of the values of A, because it covers all possible states of the node v. In fact, the constant equals to $2^{\text{size}(v)}$ (where $\text{size}(v)$ is the size of the subtree of v) because that is all the possible number of ways to assign parameters to the subtree of v.

Let's rearrange the $\texttt{dp}$ formula from subtask 2:

$$\begin{aligned}\texttt{dp}_1(u) &= \texttt{dp}_0(l)\texttt{dp}_1(r)+\texttt{dp}_1(l)\texttt{dp}_0(r)+2\texttt{dp}_1(l)\texttt{dp}_1(r)\\ &= \texttt{dp}_1(l)(\texttt{dp}_0(r)+\texttt{dp}_1(r))+\texttt{dp}_1(r)(\texttt{dp}_0(l)+\texttt{dp}_1(l))\\ &= \texttt{dp}_1(l)(2^{\text{size}(r)})+\texttt{dp}_1(r)(2^{\text{size}(l)})\end{aligned}$$

Note that similar rearrangement can be done for $\texttt{dp}_0(u)$ as well.

Another way to interpret the formula rearrangement is that: the number of ways to assign a parameter to a threshold gate is isomorphic to a multiplexer (i.e. the threshold gate can be thought as an operator that takes the value from one of its two children). Using this interpretation, we can count, for each leaf with state 1, how many ways for it to reach the root.

The number of ways for a leaf v to reach the node is $2^{N-\text{depth}(v)}$. That means when the state is 0, the leaf will contribute 0 towards the answer, and $2^{N-\text{depth}(v)}$ otherwise. The range update can be done using a segment tree that is similar to subtask 5.

Time complexity: $O(N+M+Q\log M)$

4.7 Subtask 7

We can extend the interpretation from subtask 6 to this subtask. That is, the threshold gate can be seen as an operator that takes a value from one of its children.

Therefore, for each leaf, we can mark all the paths from it to the root, and then compute the product of the number of children of all unmarked internal nodes in $O(N + M)$. The product will be the contribution value of this leaf. The update can again be performed using the help of a segment tree.

Time complexity: $O(M(N + M) + Q \log M)$

4.8 Subtask 8

Continuing from subtask 7, we can optimize the computation of the contribution value of each node. First, for each node v, we compute the product of the number of children for all the internal nodes in the subtree of v. Let the values be $subtree(v)$. This can be done in $O(N)$.

Next, we can compute the contribution of all leaves in a single DFS. When traversing to a child of a node, we need to multiply all the $subtree$ values from all the other children to the final product. To do this quickly, we can order the children $c_1, c_2, \ldots$ and for each child c, we calculate the prefix product and suffix product of $subtree(c)$. To traverse to child c_i, we multiply the product of the prefix product of $subtree(c_{i-1})$ and the suffix product of $subtree(c_{i+1})$.

The update can be done similarly as subtask 7.

Time complexity: $O(N + M + Q \log M)$

5 Rarest Insects

Written by: **Hazem Issa** (Egypt), Egyptian Olympiad in Informatics (EOI)
Prepared by: Abdul Malik Nurrokhman
Solutions, review, and other problem preparations by: Hocky Yudhiono, Jonathan Irvin Gunawan, Prabowo Djonatan
Analysis author: Abdul Malik Nurrokhman

For this discussion, let D be the number of different types of insects, and ans be the cardinality of the rarest insects.

5.1 Subtask 1

In this subtask, $N \leq 200$.

Let x be an insect whose type is unknown. For all other insects y, we can check whether they have the same type by putting them (and only them) inside the machine and then pressing the button. If the button replied with 2, that means they are both the same type.

Do it for every pair of insects x and y to obtain the answer.

Query complexity: $O(N^2)$

We can do further optimization by ignoring some insects whose types are already known.

Query complexity: $O(N \times D)$

5.2 Subtask 2

In this subtask, $N \leq 1000$.

Let I be the number of insects inside the machine.

To know the value of D (the number of different types of insects), we need to put all insects one by one and press the button each time. If the machine reports one more than the previous press, then move the most recently inserted insect outside. When this is done, $D = I$.

Repeat the above process again for insects which are outside the machine. This time, our goal is to check whether the rarest cardinality is 2. If $I < 2D$ after this second process, then the answer is 1. Otherwise, repeat the process again to check whether the answer is $2, 3, \ldots$.

Query complexity: $O(N \times ans)$

Note that $ans \times D \leq N$, so $\min(ans, D) \leq \sqrt{N}$. If we combine the last two solutions above, then we can solve this subtask.

Query complexity: $O(N\sqrt{N})$

5.3 Subtask 3

In this subtask, $N \leq 2000$ and a partial score might be given.

5.3.1 $O(N \log N)$ query complexity

Let B be the upper bound of the answer. That means B is initially N.

To check whether b can be the bound, we insert the insects one by one, and move the recently inserted insect outside again whenever the press button replied with $> b$. If $I = b \times D$ after this process, then $ans \geq b$. We will call this check the *upper bound check*.

We can binary search the upper bound so we can get around 50 points.

5.3.2 $O(N \times \textbf{invfact}(N))$ query complexity

Let invfact(N) be the inverse factorial function.

We can initially set $B = \frac{N}{D}$. If $I = B \times D$ after each upper bound check, then the answer is B, else set $N = I$ and repeat the process. With this trick, we only need to check invfact(N) upper bounds. We will get around 70 points with this solution idea.

5.3.3 $O(3N)$ query complexity

From the binary search solution, we don't exactly need to re-empty the machine. After each iteration, if $I = B \times D$ then keep the insects inside the machine without removing them. Otherwise, if $I < B \times D$ then just remove all insects that are inserted in the last iteration. Therefore, the maximum operations needed for each type are divided by two for each iteration. Note that, $N + \frac{N}{2} + \frac{N}{4} + \cdots = 2N$. We also need an additional N operations in the beginning to get the value of D.

5.3.4 Constant optimization

The previous solution actually yields slightly more than $3N$ so we need further optimization. One of the optimizations that we can do is to stop moving insects inside the machine when $I = B \times D$.

6 Thousands Islands

Written by: **Félix Moreno Peñarrubia** (Spain), Universitat Polit'ecnica de Catalunya - BarcelonaTech
Prepared by: Alham Fikri Aji
Solutions, review, and other problem preparations by: Jonathan Irvin Gunawan, Maximilianus Maria Kolbe Lie, Muhammad Ayaz Dzulfikar, Prabowo Djonatan
Analysis author: Maximilianus Maria Kolbe Lie

Definition 6.1. *A canoe whose state is currently docked at island u and can be sailed to island v is defined as (u, v).*

Definition 6.2. *Suppose that canoe i is in the original state, or $(U[i], V[i])$. Define i' as the reversed state (after being sailed for an odd number of times), or $(V[i], U[i])$.*

6.1 Subtask 1

In this subtask, $N = 2$.

Note that each canoe should be sailed an even number of times. For $N = 2$, at least two $(0, 1)$ canoes and one $(1, 0)$ canoe are required for a valid journey. Suppose that $c_1 = c_2 = (0, 1)$ and $c_3 = (1, 0)$. One of the valid journey is $[c_1, c_3, c_2, c'_1, c'_3, c'_2]$.

Time complexity: $O(M)$

6.2 Subtask 2

In this subtask, $N \leq 400$ and the original state of the canoes is guaranteed to be a complete graph.

For $N = 2$, the canoes do not fulfil the minimum requirement for a valid journey, as mentioned in Subtask 1. For $N \geq 3$, a valid journey can be constructed as follows. Suppose that $c_1 = (0, 1)$, $c_2 = (1, 0)$, $c_3 = (0, 2)$, and $c_4 = (2, 0)$. One of the valid journey is $[c_1, c_2, c_3, c_4, c'_2, c'_1, c'_4, c'_3]$.

Time complexity: $O(M)$

6.3 Subtask 3

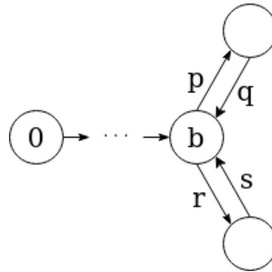
In this subtask, $N \leq 1000$ and the original state of the canoes is guaranteed to be a bidirectional graph.

Observation 6.1. *If an island u has only 1 docked canoe to island v, then a valid journey from island u (to island u) exists if and only if a valid journey from island v (to island v) exists. Therefore, island u can be removed. Instead, find a valid journey from island v.*

Proof. For a canoe $c = (u, v)$, after reaching island u, use canoe c to visit island v. Complete the journey and go back to island v. Use canoe c' to go back to island u. Hence, a valid journey from island u exists. ☐

First, apply Observation 6.1 starting from island 0 until no island satisfies the observation.

Definition 6.3. *Denote b as the first island from island 0 that does not satisfy Observation 6.1. Refer to the following illustration for more clarity.*



If island b does not exist (the graph is a line graph), then a valid journey does not exist as well. Otherwise, the following is one of the valid journeys from island b. After reaching island b, use the following path: $[p, q, r, s, q', p', s', r']$, which will lead back to island b. By Observation 6.1, a valid journey also exists from island 0. Note that canoe p and r might sail to the same island.

Time complexity: $O(N + M)$

6.4 Subtask 4

In this subtask, $N \leq 1000$ and each edge has a duplicate.

Observation 6.2. *Islands with no docked canoes can be removed without changing the answer. Formally, such islands are nodes with* 0 *out-degree.*

Proof. Using proof by contradiction, if an island with 0 out-degree is part of a valid journey, then the journey will never be a tour, since it is impossible to get out from such an island. ☐

Note that all canoes which can be sailed to removed islands can be removed as well. The action of removing such islands might result in new islands with no docked canoes, which can be removed too. These operations can be achieved in $O(N + M)$ using DFS until no island can be removed.

The construction in Subtask 3 can be generalized to finding two cycles that can be visited from island b. Some canoes, or none, might be sailed from island b to each cycle. This generalization is correct in Subtask 3, as the edges are bidirectional.

Due to the constraint of this subtask, finding one cycle will automatically find the other cycle. Therefore, a valid journey exists if a cycle can be found in the graph. If no cycle exists in the graph, then island 0 will be removed by Observation 6.2, which implies no valid journey exists. The cycle finding can be achieved in $O(N + M)$ using DFS.

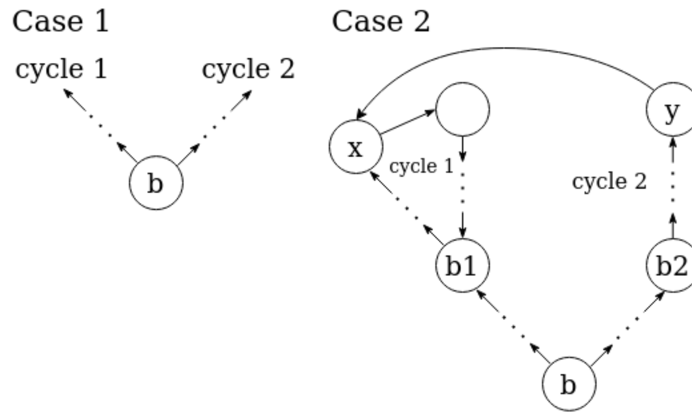
Time complexity: $O(N + M)$

6.5 Subtask 5

To solve the full problem, we shall use some observations and constructions similar to those from the previous subtasks. However, the algorithm that removes islands in Observation 6.1 and 6.2 should be done simultaneously. Keep removing the islands until no island satisfies either observation.

Next, two valid cycles should be found from island b. Note that the cycle refers to a set of canoes, not a set of islands. It is possible to visit the same islands, as long as it uses different canoes.

After reaching island b, there are 2 different cases that need to be handled. These cases are easier to be described by the following illustrations. Note that the dots in the illustration refer to a list of canoes, which can consist of zero or more canoes.



Case 1: all canoes used in both cycles are disjoint. Using the similar construction in Subtask 4, a valid journey exists in this case.

Case 2: there exist the same canoes in both cycles. The following construction satisfies this case. After reaching island b, go to island b_1. Finish the first cycle, then go back from island b_1 to island b. Go to island b_2 and traverse the second cycle until island x, which is a part of the first cycle. Finish traversing the first cycle in reverse from island x. Then, from island x, go back to island y, and finish traversing the second cycle in reverse until island b. Two cycles are completed, which implies a valid journey exists in this case.

Time complexity: $O(N + M)$