Mock INOI 7 Solutions

roych9863

November 2021

1 General Comments

This mock was indeed very difficult. However, it is by no means impossible. There were many simple subtasks given for contestants to solve. Most important in a difficult contest like this is to keep calm and take the easy subtasks before pushing for some of the more difficult but still achievable subtasks if you have time.

The expected difficulty of the questions was (from easiest to hardest) B, C, A. The cutoffs we decided for this mock were:

- Silver (Intermediate):
  - A: 30
  - B: 56
  - C: 35
  - Total: 121
- Gold (Advanced):
  - A: 30
  - B: 100 (full solve)
  - C: 67
  - Total: 197

This mock had many subtasks and so there were other ways you could come to these scores. For example, a gold-scoring participant may get 18 more points for subtask 6 on A or 98 points on C, whereas a silver-scoring participant may get 32 more points for subtasks 5 and 6 on C, or 8 more points for subtask 6 on B.

The skills and knowledge expected for participants of each cutoff is the following:

- Silver:
  - A: Statement-reading and observation

  - B: Brute force, segment tree, and dynamic programming
  - C: Brute force and implementation
- Gold:
  - A: Statement-reading and observation
  - B: Observation and segment tree
  - C: Brute force, implementation, and greedy
- AC:
  - A: Observation and depth-first search
  - B: Observation and segment tree
  - C: Greedy, dynamic programming, and observation

This mock provided many opportunities for subtasks. Unfortunately, many were overwhelmed by the difficulty of this mock and unable to take these subtasks. We hope that all participants take this mock as a lesson on contest strategy.

2 A. Little Garden

This is (by far) the hardest problem of the set. There are some subtasks which are more approachable than others, however. If you skip this problem due to the long statement, you are missing out on 30 easy points.

2.1 Subtask 1

This can be found from the sample or from the statement. A tree with only one node can be replaced by any other tree. Therefore, the answer is always "Almost Complete".

2.2 Subtask 2

If we have a single-node tree, then the answer clearly must be "Almost Complete" again. We focus on the case where all trees have height 2. There are 3 possible trees like this. It can be observed from the samples and also from thinking that we need all 3 of these trees to be in the input for the answer to be "Almost Complete"; otherwise, the answer is "No".

2.3 Subtask 3

Lemma 1: If we can construct all trees of height $\max h$, the answer is "Almost Complete"; else, the answer is "No".

After we discover this lemma, we can brute force to see which trees of height $\max h$ we can construct.

2.4 Subtask 4

Lemma 2: Consider trees T, T' such that T can grow into T'. If we are able to construct T, we can construct T' as well.

Define a "branch" as a chain, plus possibly the sons of some nodes on the chain. In other words, all non-leaf nodes of the tree have either one or two children. If a node has two children, the subtree of one of the children is a chain, and the subtree of the other is a single node.

Lemma 3: We can get a "basis set" of trees such that from this basis set we are able to generate every other tree. This basis set is the set of all branches.

As a result of Lemmas 2 and 3, observe that we can disregard all trees in the input which are not branches.

Theorem 4: If we are able to generate all branches of height $\max h + 1$, the set is Almost Complete.

From Theorem 4, we fairly directly get a $O(4^h)$ approach to solve this problem (note that the number of branches of height $h + 1$ is $O(4^h)$).

2.5 Subtask 5

In fact, the only way to get "Almost Complete" is for the single-node "chain" to appear in the input. Consider a set θ consisting of all chains except the single-node "chain". This set is still not almost complete - every tree with nodes in both the left subtree and right subtree of the root cannot be generated.

2.6 Subtask 6

This subtask can be considered an easier version of the general problem. Let's start from the root. Of course, if the tree with one node appears then the answer is "Almost Complete". We disregard that case. If all three trees of height 2 appear, then the answer is again "Almost Complete". However, if the complete binary tree of height 2 does not appear, then the answer is "No". In this way, we can formulate a recursive strategy. For a node v with children l and r, we consider all trees in the input who have leaves l and/or r. If the tree with both leaves l and r does not exist, we directly output "No". If all three trees (l, r leaves; only l leaf; only r leaf) exist, we stop recursing. If the l-leaf and/or r-leaf trees do not exist, we recurse down to that node.

2.7 Subtask 7

From earlier, discard all trees in the input that are not branches. Now, consider the lemmas from subtask 4. We can represent a branch of length h as a pair (c, l), where $|c| = h - 1$ records the form of the main chain and $|l| = h + 1$ records the existence of leaves on the main chain from top to bottom. These can be called "main chain code" and "leaf code" respectively. For convenience, we can use 0 to denote Left Child/Leaf Does Not Exist and 1 to denote Right Child/Leaf Exists. Note that the last two elements of the leaf code represent

left child and right child respectively since the last node on the chain may have two children.

Theorem 5: Let the representation of branch C be (c, l) where $c = c_1c_2 \dots c_{h-1}$, $l = l_1l_2 \dots l_{h-1}xy$. The branches of length $k < h$, (c', l') generate C if and only if the following two conditions are met:

- $c' = c_1c_2 \dots c_k$
- $l' = l_1l_2 \dots l_{k-1}x'y'$. If the $k+1$-th node on C is the left child of the k-th node, then $x' = 1, y' = l_k$, otherwise $x' = l_k, y' = 1$.

For a set of branches $\alpha = (c^1, l^1), \dots (c^n, l^n)$, consider building a trie T_1 on $c^1 \dots c^n$ and a trie T_2 on $l^1 \dots l^n$. We now attempt to determine if there is a branch (c, l) of length h that cannot be grown. Consider the following method: DFS on T_1, and maintain generated leaf codes on T_2 (specifically, for each leaf code in T_2, maintain the number of different branches that can generate it). When the DFS reaches a node u which holds one of our original branches, update the nodes in T_2 which correspond to leaf codes that this branch can generate. When we leave the subtree, reverse these updates. When we reach the "outside" of T_1 (i.e. reach a leaf) we then verify to see if there are leaf codes in T_2 that cannot be generated. The time complexity is $O(|\alpha||T_2|) = O(mn)$.

2.8 Subtask 8

Note that our updates correspond to a subtree of T_2. Therefore, we maintain a segment tree that can do range increments and decrements on a subtree and can query for the minimum value. If the minimum value is 0, that means there is some leaf code which could not be generated. The complexity is $O(|\alpha| \log |T_2|) = O(m \log n)$.

2.9 Subtask 9

Let's consider branches from a different perspective. Specifically, let's create a recursive, formal definition of branches.

Definition: Consider the subtrees l and r of the children of root v. A branch rooted on v satisfies one of the following 4 requirements:

1. l is empty, r is a branch.
2. r is empty, l is a branch.
3. $|l| = 1$, r is a branch.
4. $|r| = 1$, l is a branch.

(We can note that cases 3 and 4 may overlap, but this is not a major concern.)

Similar to the previous subtask, we can construct a list $s = c_1c_2 \dots c_{h-1}$ for each branch, where c_i is a value in $[1, 4]$ corresponding to the state of the child subtrees of the main chain node on the branch level at i. After merging all

branches together we get a quadtree T_3 (this can be considered similar to the binary tries in Subtask 7 and 8).

Define a node u on T_3 as legal if there is only a limited number of trees that cannot be generated from subtrees of u. How do we check if a node u is a legal node? First, if it is a leaf node, u is obviously legal (this corresponds to the "single-node" case from before). What if u isn't a leaf node? Check u's **four children** v_1, v_2, v_3, v_4. Solve them separately to determine whether they are all legal; if they are all legal, the subtree is legal; otherwise the subtree is illegal. This recursion can all be accomplished with simple DFS.

Note that the number of nodes in the quadtree is on the same order as the input, and the complexity of divide and conquer is on the same order as the number of nodes in the quadtree, so the final complexity of this solution is linear.

If you are confused about the solution, you may wish to compare this solution to the recursion for Subtask 6 and the construction of tries for Subtask 7.

3 B. Lazy Setter

This is the easiest problem of the set. However, it is by no means easy! With a simple insight, you can instantly solve the problem, or you can engage with the somewhat standard subtasks (a well-prepared contestant is expected to get all subtasks except the last one).

The problem is dedicated to lxl (Li Xinlong, ODT), who is the inventor (or at least populariser) of the solution method and has written many problems regarding sqrt decomposition and range inversion queries.

3.1 Subtask 1

An inversion involves at least 2 elements, so if $n = 1$, there can be no inversions. Output 0 for each query.

3.2 Subtask 2

This subtask is for simple brute force solutions. For each query, we can directly check all pairs (i, j) such that $l \leq i < j \leq r$ in $O(n^2)$. For each update, we can directly update the values of a_i in $O(n)$. The final complexity is $O(mn^2)$.

3.3 Subtask 3

It is well-known that we can solve for the number of inversions in an array in $O(n \log n)$. One way is by optimising the previous method with a Segment Tree. Another way is by using merge sort to count the number of inversions. The final complexity is $O(mn \log n)$.

Note: The limits for this subtask are quite high because the constant factor of Subtask 2's solutions is very low and can plausibly pass this subtask with lower limits.

3.4 Subtask 4

Since there are no queries, our code need not output anything. Submitting an empty program will pass this subtask.

3.5 Subtask 5

Since there are no updates and the value of n is small, we can precompute the answer for a query on $[l, r]$ in $O(n^2)$ and use these precomputed values to directly answer each query in $O(1)$. We do this with dynamic programming. Let $inv[l][r]$ be the number of inversions between the values in $[l, r-1]$ and a_r. For each r, we iterate in descending order of l, and calculate this in $O(n^2)$. Now let $dp[l][r]$ be the number of inversions in the interval $[l, r]$. It is clear that $dp[l][r] = dp[l][r-1] + inv[l][r]$. Therefore, our final time complexity is $O(n^2 + m)$.

Note: Solutions for subtask 6 can pass this subtask as well. This decision was made since it is impossible to prevent optimised Mo's solutions from passing this subtask even if we set $m = 2 \times 10^5$, so we allowed for all of them to pass here.

3.6 Subtask 6

This subtask is for solutions using Mo's Algorithm. With segment tree, we can update the answer in $O(\log n)$ for each movement of the right or left pointer, and so we achieve complexity $O(n\sqrt{n}\log n)$, where $m = O(n)$.

Note: The limits for this subtask are low because solutions using recursive segment tree with Mo's Algorithm run quite slowly.

3.7 Subtask 7

This is the full solution, utilising the condition that the testdata is randomised.

The intuition for this solution is that due to the range set updates, the number of unique values in the array decreases quite rapidly. Therefore, we can simply store intervals with the same value using std::set or std::map, and brute force walk over these intervals for our updates and queries. The remainder of the solution is similar to subtask 3.

3.8 Proof of Complexity

This proof is rough and omits technical details.

Suppose we have a randomly-selected range $[l, r]$ which contains x intervals. With 1/2 probability (query), we use $O(x)$ time to brute force walk over the intervals, and delete no intervals. On the other hand, with 1/2 probability (update), we use $O(x)$ time to brute force walk over the intervals, and delete $O(x)$ intervals. As a result, the complexity of the brute force walks is linear.

The remaining part of the complexity is the predecessor search - i.e., finding the first interval in our brute force walk. This works in $O(\log t)$, where t is the

total number of intervals at the time. We can trivially bound t on $O(n)$ for $O(n \log n)$. In fact, we can prove that this works in $O(n \log \log n)$ as well, but in this problem we utilise segment tree so the overall complexity is $O(n \log n)$ regardless.

4 C. Peaceful Times

This problem first requires the skill of greedy to solve the $m = n - 1$ case and dynamic programming to come to a full solution. On the other hand, if you are unable to solve the $m = n - 1$ case, there are 4 simpler subtasks given for you to solve.

4.1 Subtask 1

There is only one box. If there is ≤ 2 colours, we can place all the balls into this box. Otherwise, it is impossible to find an answer.

4.2 Subtask 2

This subtask can be solved by brute force in $O(2^{10})$. We consider all individual balls separately and try all combinations of balls in the boxes.

4.3 Subtask 3

The solution from the previous subtask can be generalised to work in $O(m^{m*k})$. In this case, this complexity is $O(3^{15})$. The key is instead of using base-2 conversion to find the combination, we use base-k conversion, in this case base-3 conversion.

4.4 Subtask 4

Since each box has size 2 and we are allowed to place 2 colours in a box, it is always possible to find a solution. For each individual ball, just allocate it to the next available box.

4.5 Subtask 5

This is the first serious subtask of this problem. In this case, it is always possible to find a solution.

The following greedy works: for each box, take the colour with the smallest number of balls remaining, and pair it with the colour with the largest number of balls remaining. The idea is that with each box we reduce the number of colours remaining by 1 until we are left with 1 box with 2 colours remaining. This can be proven by Pigeonhole Principle (which can be used to inspire the solution.)

4.6 Subtask 6

We can construct the solution for this subtask from the previous subtask. We can reduce the number of boxes by 1 without reducing the number of colours repeatedly by taking the colour with the largest number of balls remaining repeatedly. By Pigeonhole, this is always possible. Then we get into a situation where $m = n - 1$ again and we solve using the same solution as Subtask 5.

4.7 Subtask 7

We are left with the hard case where $m = n-2$. Let's consider the graph where if each box has colours u and v, there is an edge between u and v. (If the box has only one colour u, this can be represented as a self-edge (u, u).) Clearly, the number of edges is m and the number of nodes is n. Now, since $m < n - 1$, the graph must be disconnected. In addition, at least one of the connected components must be a tree. This connected component can be solved with the $m = n-1$ case. In addition, the remainder of the graph also satisfies $m = n-1$, so it can also be solved by the $m = n-1$ case. We note that this is only possible if $\sum_{i \in C} b_i = k \times (|C| - 1)$, where C is the connected component. Therefore, we need to find some set C which satisfies this. We can do this with dynamic programming. Let $dp(i, j, k)$ be a boolean which represents if it is possible to form sum j from k elements of the first i elements. This DP runs in $O(n^3k)$.

4.8 Subtask 8

We seek to optimise the previous solution. Note that if we let $b_i := b_i - k$, we only need to find a sum of $-k$ and no longer need the state to contain the number of taken elements. Thus, we can have $dp(i, j)$ denoting whether it is possible to form modified sum j from the first i elements. The final complexity is $O(n^2k)$.

4.9 Subtask 9

By using bitset optimisation, we can improve complexity by a factor of 32 or 64. The final complexity is $O(\frac{n^2k}{w})$.