IOI 2021 MYS Sparring Contest Solutions

Zi Song Yeoh

June 4, 2021

1 Rock-Paper-Scissors Tournament

First, we need to know how to compute the values of A, B, C given a fixed string s. If we know the type of every person (i.e. there are no question marks), then it's easy since we can just simulate the entire tournament. The same method also allows us to calculate the values of A, B, C.

1.1 Computing A, B, C given an output

Let's construct a tournament tree, where the leaves are the participants of the first round and the parent of two nodes is the winner of the match played between the children. The root is the winner of the tournament.

For each node of the tree, compute the number of ways to fill the question marks in the subtree so that the resulting player in the node is R, S, and P respectively. For the leaves, if the player is of type R, then the value is $(1, 0, 0)$. If the player is of type S, the value is $(0, 1, 0)$. If the player is of type P, the value is $(0, 0, 1)$. Finally, if the player type is unknown (i.e. a question mark), the value is $(1, 1, 1)$.

We can compute the value of a node (c_0, c_1, c_2) given the values of its two children (a_0, a_1, a_2) and (b_0, b_1, b_2). This can be done in $O(1)$ by noting that $c_0 = (a_0 + a_1)(b_0 + b_1) - a_1 b_1$, i.e. the number of ways to get a player of type R in this node is the number of ways for both children to be of type S or R and not both of them can be of type S. Similar identities hold for other c_i. Thus, we can compute the values of all nodes while building the tree (in $O(N)$ time, where $N = 2^n$ is the number of players). Finally, the values in the root of the tree corresponds to the desired A, B, C.

1.2 Back to the original problem

Now, we need to solve the inverse problem : Given A, B, C we have to find a string s such that the tournament tree produced by s has root (A, B, C). Note that the problem statement states that the string s can contain at most 32 characters, i.e. there are at most 32 players. This number looks small so maybe we can do something about it.

What if we try to enumerate all the possible tournaments? Let's count how many such tournaments are there. There are 4 ways to choose each player (R, S, P, ?), so this means there are 4^{32} tournaments with 32 participants alone, which is way too much to be enumerated. Note that simply brute forcing all tournaments should work for the subtasks with $|s| \leq 8$.

Key Observation: Note that the only important thing is the value of (A, B, C) produced by the tournament. It turns out that this reduces our search space dramatically. Let's start from the 4 tournaments of size 1 each producing a different triple. To generate all distinct triples of size 2^n, we take all possible pairs of triples of size 2^{n-1} and combine them (i.e. concatenate them and calculate the value, which can be done in $O(1)$ by just combining the two triples like how we compute the value of node with the value of its children). Then, we can use a map that maps a triple to a string to store the possible triples formed from strings of size 2^n and the string that generates the triple.

Using this method, we find that there are 23121 possible triples generated from tournaments of size 16. Thus, by storing all the triples formed by tournaments of size $1, 2, 4, 8, 16$ as well as the tournament that generated them we can answer the question for a triple (A, B, C) if it's generated from a string of length ≤ 16. This will solve the subtasks with $|s| \leq 16$.

1.3 Full Solution

However, finding all possible triples from combining two tournaments of size 32 might take long as we need to try all pairs of triples (and even if we did find all of them in time we don't have the memory to store every distinct triple). So, how do we know if a triple (A, B, C) is produced by a tournament of size 32?

A tournament of size 32 is formed by combining two tournaments of size 16. Let's iterate over all possible triples for the first tournament of size 16, (a_0, a_1, a_2). We need to determine if there exist a triple from a tournament of size 16, (b_0, b_1, b_2) such that their combination is (A, B, C).

We have the equations $b_0(a_0+a_1)+b_1a_0 = A, b_1(a_1+a_2)+b_2a_1 = B, b_2(a_2+a_0)+b_0a_2 = C$. These are just simultaneous equations in b_0, b_1, b_2 so we can solve them. However, we need to be careful when some of the elements are 0 and also the fact that the intermediate values might exceed the range of long long.

Verify that if $a_0 \neq 0, a_1 \neq 0, a_2 \neq 0$, we have $b_0 = \frac{\frac{A(a_1+a_2)}{a_0}+\frac{Ca_1}{a_0+a_2}-B}{\frac{(a_0+a_1)(a_1+a_2)}{a_0}+\frac{a_1a_2}{a_0+a_2}}$ and similar identities.

For example, we can use long double to store the intermediate results and in the end, we try close integer values in case of precision errors for each value of b_i. We check whether our triple (b_0, b_1, b_2) works by combining it with (a_0, a_1, a_2) to see if it produces (A, B, C) and also check if it's among the triples produced by tournaments of size 16.

Since we only loop through 23121 triples per query, and for each triple we use $O(1)$ time to check if it works (albeit with a slightly large constant), this is sufficient to pass.

2 Tree Subset Diameter

The subtask with $N \leq 16$ can be solved by merely brute forcing all subsets and calculating the diameter of the set naively. We need more insights to obtain a polynomial-time solution.

2.1 Quadratic-time Solution

First, let's find any polynomial-time algorithm. The key idea (which is also somewhat hinted by the subtasks D even, D odd) is to consider the center of the spanning tree spanned by our subset. Recall that a tree has either one or two adjacent centers. Hence, if we fix a center (or a central edge), we can find an easy description of the set of vertices we can choose such that the diameter is D and the center(s) is/are the fixed vertex/vertices.

For example, suppose D is even. Then, there must be a unique center (midpoint of the diameter). Fix vertex v as the center. Set S is valid if and only if

- All chosen vertices are at distance $\leq \frac{D}{2}$ from v.
- There are at least two chosen vertices at distance $\frac{D}{2}$ from v such that v lies on the path connecting them.

Root the tree at v and let the i-th children of v have exactly c_i vertices at distance $\frac{D}{2}$ from v in its subtree. Let k be the number of children of v. Then, the number of ways to choose S such that the center is v and diameter is D is exactly $2^a(2^{c_1+c_2+\cdots+c_k} - (2^{c_1} - 1) - \cdots - (2^{c_k} - 1) - 1)$ where a is the number of vertices at distance $< \frac{D}{2}$ from v. We can compute the desired values using a naive DFS from v. Hence, we can process each vertex v in $O(n)$ time. This immediately gives a $O(n^2)$ solution.

The case D is odd is analogous, except now we fix an edge as the central edge instead of fixing a vertex.

2.2 Subquadratic time solution

The key hurdle of this problem is to obtain the values $c_1, c_2, \cdots, c_k$ as defined in Section 2.1 fast enough. Root the tree at vertex 1. In fact, it is sufficient to precompute the following for $d \in \{D/2 - 2, D/2 - 1, D/2\}$ and all v:

- The number of vertices at distance $\leq d$ from vertex v.
- The number of vertices at distance $\leq d$ from vertex v **in the subtree rooted at** v.

The former can be computed in $O(n \log n)$ time by centroid decomposition. The latter can be computed by merging arrays on tree in $O(n \log n)$ time (using small-to-large trick). Hence, we can obtain a $O(n \log n)$ solution (with albeit large constant factor).

3 Class Division

Firstly, suppose the class sizes are fixed (e.g. when $l_i = r_i$). Then, it is optimal to place the highest scoring students in the smallest class (proof by rearrangement inequality/exchange argument). Hence, we can sort the students in decreasing order of scores, and once we know the sizes of each class we can compute the total score of the school in $O(m)$ time.

With this observation alone, we can solve the subtask $l_i = r_i$ and $m \leq 4$ (by brute forcing the class sizes). There is also a subtask-specific solution for $l_i = 1$ and more generally $l_i \leq l_{i+1}, r_i \leq r_{i+1}$ but I will not discuss it here.

3.1 Key Observation

Call a multiset of class sizes $(s_1, s_2, \cdots, s_m)$ (sorted in nondecreasing order) *valid* if we can choose the multiset of class size. Suppose there exists some $i < j$ such that $(s_1, s_2, \cdots, s_{i-1}, s_i - 1, \cdots, s_{j-1}, s_j + 1, \cdots, s_m)$ is valid. Then, it does not hurt us to choose this new multiset of class sizes (you can verify that the score of every class does not decrease). Call the transformation $(s_1, s_2, \cdots, s_m) \to (s_1, s_2, \cdots, s_{i-1}, s_i - 1, \cdots, s_{j-1}, s_j + 1, \cdots, s_m)$ a *swap*.

Now, let S be the **lexicographically smallest** optimal multiset of class sizes (where the elements of our multiset is sorted in nondecreasing order). The observation above shows that S cannot have any *swaps*.

Call a class i *special* if we choose its class size to be some integer in (l_i, r_i). We can show that there is at most one *special* class in S by noting that if there were two *special* classes, we can always perform a swap.

3.2 Exponential-time Solution

Since there is at most one *special* class, we can try all classes and fix them as the *special* class (or it may happen that all classes are non-special). We can then try all 2^{m-1} possibilities of assigning the sizes of the remaining classes (since each remaining class must have either class size l_j or r_j). For each possibility, we can check it in $O(m)$ time. Hence, this gives an immediate $\tilde{O}(2^m \cdot m)$ time solution.

3.3 Polynomial-time Solution

To obtain a polynomial-time solution, we need to further exploit the problem structure. Again, let us fix the *special* class C and see how we should choose the sizes of the remaining classes. From now on, we will ignore the existence of class C unless otherwise specified. Call a class an L-type if we decide to assign its class size as l_i and R-type otherwise. If $l_i = r_i$ for some class, we consider it as a L-type class.

Let $A = [l, r]$ be the R-type class with the **minimal** l, and among those, with minimal r. By definition, $l < r$. If no such class exists, all classes are of L-type and we can check this separately. Here are a few important observations:

- If some other class $A' = [l', r']$ has $l' < l$ or $l' = l$ and $r' < r$, then it is a L-type by definition. Also, there might be multiple copies of $[l, r]$ in the input. In this case, we also identify each class with its index and those with smaller indices are automatically of L-type while those with larger ones are of R-type.
- If some other class $A' = [l', r']$ has $l' \geq l$ and $r' \geq r$ and $(l', r') \neq (l, r)$, then we claim that either $l' = r'$ (for which it doesn't matter what type it is) or it must be an R-type. If $l' \geq r$, we can perform a swap directly. Otherwise, we can let class A have size l' and class B have size r without changing the result. However, this forces at least one of A or A' to be *special*, a contradiction.

In particular, once we fix A, we automatically know the sizes of all segments that do not lie strictly inside A! Now, it seems that we can recursively solve the problem on the segments which lie strictly inside A. More specifically, we can use some dynamic programming with states $dp[i][L][R]$ which denotes the maximum score if we only care about segments lying strictly within segment i and that we will assign the L-th to R-th students to these classes. However, we have to remember to take the class C which we excluded as *special* into account, because the relative position of its class size will affect which students are assigned to which classes. Fortunately, it turns out that we can incorporate that into our dp thanks to the following observation.

Lemma 3.1 *Suppose class C is special and has size s. Then, s lies (non-strictly) between the largest size of a L-type class and the smallest size of a R-type class among the classes not lying strictly within A. In other words, we can assign students to classes not lying strictly within A greedily without caring about class C.*

The lemma can be proven by again abusing the fact that we cannot have swaps in a lexicographcially minimal optimal solution.

Armed with this fact, we can now safely compute $dp[i][L][R]$. The transitions are done by iterating through the "leftmost" R-type class remaining and using the same pattern above to reduce to a smaller segment. With proper precomputation, we can do the transitions in $O(m)$ time (or more precisely, $O($number of segments strictly inside segment $i))$. Hence, this gives a $O(m^3n^2)$ solution ($O(m)$ for iterating over all possible C, $O(m^2n^2)$ for this dp), though in practice constant factor is much smaller since a lot of states are actually unvisited (in fact, the author's solution runs in less than 200ms).