ACM-ICPC Dhaka Regional 2017 Preliminary Editorial

Problem A (Germ Control):

Setter: Aninda Majumder
Alternate Writer: Md. Samiul Islam

Setter Solution:
This is a variation of parentheses balancing and can be solved with a straightforward non-lazy segment tree.

We can imagine replacing **L** and **R** with **(** and **)** respectively. So the problem can be easily thought of checking if the parentheses are balanced.

In each node, we keep an imbalanced parentheses imagining the characters as consecutive **)**s followed by consecutive **(**s, i.e. **))))((((**.

Also, we can imagine each **#** as **()**, i.e. a single balanced parentheses and always consecutive **#**s will take 1 day to dissolve together. Note, that we don't have to actually replace the **#**s with **()**, they will be handled in the segment tree automatically.

In each node we will have the following information:

- Number of imbalanced **R**s, **numR**.
- Number of imbalanced **L**s, **numL**.
- Minimum time when the rightmost **R** dissolves so far, **minTimeR**.
- Minimum time when the leftmost **L** dissolves so far, **minTimeL**.
- Overall minimum taken so far for this node, **minTime**.

Update:

For each single character update, we go to the leaf and update accordingly and merge the nodes towards the root.

Computation at the leaf node:

At the leaf, we update the number of imbalanced characters with 1 and also update the minimum time so far to dissolved them with 1 trivially.

- Initialize **numR** or **numL** to 1 if it is **R** or **L** respectively.

- Initialize **minTimeR** or **minTimeL** to 1 if it is **R** or **L** respectively.
- Initialize **minTime** to **1**. Note that **#** will be automatically handled here.

Merging the left and right segments:

If we have a left segment and right segment representation as **RRRLLL** and **RRRRRLLLL** respectively, merging them we'll have:

RRRLLLRRRRRLLLL -> **RRR[LLLRRR]RRLLLL** -> **RRRRRLLLL**

- We find the minimum of **left segment R** and **right segment L**, **minNum = min(numR, numL)**. So, **minNum * 2** will be the length of the currently balanced parentheses after merging the nodes.
- Update **numR = left segment numR + right segment numR - minNum** and same way, update **numL = right segment numL + left segment numL - minNum**. Doing this will remove the newly balanced **LL..RR** from the node. For the above example, **minNum = 3**.
- Now, update **minTimeR**, **minTimeL** and **minTime** as follows:
    - If **left segment numL < right segment numR**

      This means that after merging and balancing, we'll have extra imbalanced **R**s on the right, which will add to the imbalanced **R**s on the left since all of the previously imbalanced **L**s on the left have already been balanced. So, we can update **minTimeR** from the existing **right segment minTimeR** and from the **left segment minTime** plus the number of newly added imbalanced **R**s from the right. Also, **minTimeL** can be directly updated from the **right segment minTimeL** since all the **L**s on the left will be balanced when merging. Look at the above example for visualization. I.e.:

        - **minTimeR = max(right segment minTimeR, left segment minTime + right segment numR - minNum)**
        - **minTimeL = right segment minTimeL**.
    - If **right segment numR < left segment numL**

      We can handle this the exact opposite way as previously:

        - **minTimeR = left segment minTimeR.**
        - **minTimeL = max(left segment minTimeL, right segment minTime + left segment numL - minNum)**
    - Else, when **numL == numR**, trivially:
        - **minTimeR = left segment minTimeR**.
        - **minTimeL = right segment minTimeL**.
- Finally we update **minTime = max(left segment minTime, right segment minTime, minTimeR, minTimeL)**

Query:

When we have a query, we just check the root node. There are two cases:

1. If there is any **L**s or **R**s remaining, i.e. **root numR** !=0 or **root numL** != 0, then it's not balanced.
2. Otherwise, the answer will be **root minTime**.

Note: This method only requires the update method of the segment tree. No query method is required at all, as we only need to look at the root information.

Complexity: The overall complexity will be **O(Q * log(|S|))**, where **Q** is the number of the queries and **|S|** is the length of the string.

Alternate Solution:
We can also solve the problem using a different technique using lazy propagation on a segment tree. As we already knew from the above discussion **L** is equivalent to **(**, **R** is equivalent to **)** and **#** is **(),** now, the only thing we have to check what is the maximum depth of the bracket sequence and if the sequence is balanced.

In the given sequence, every position can be replaced with a **#** as an update. So, we can keep two indices against every index of the original sequence.
Let's consider this example: **LLRLLR#RR**

We can convert the sequence like below:
Actual sequence:

0	1	2	3	4	5	6	7	8
L	L	R	L	L	R	#	R	R

Our converted sequence:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
(		(			)	(		(			)	(	)		)		)

So, you can realize, we are taking two indices against each of the index of the sequence. If the index **i** (0<= **i** < len(sequence)) of original sequence contains **L,** we will put **(** at index **2*i** of our own sequence, if **i** contains **R**, then we will put **)** at index **2*i + 1** of our sequence. If the **i-th** character is **#**, we put both **(** and **)** at **2*i** and **2*i + 1**.
Now, we have to find the depth of the bracket sequence. Consider **(** equal to **+1** and **)** equal to **-1** and do a cumulative sum. So, we can find something like below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
(		(			)	(		(			)	(	)		)		)
1	1	2	2	2	1	2	2	3	3	3	2	3	2	2	1	1	0

Now, from the cumulative summation, we can observe that, the depth of the bracket sequence is equal to the maximum value of the cumulative sum array.

And how to find if the bracket sequence is valid? If any index of the cumulative sum array contains negative value (less than 0), then the the bracket sequence is not valid (or imbalanced). Is a sequence always valid when there is no negative sum found? The answer is no.

See the example below:
(()
The cumulative sum array will be: 1,2,1.
So, if the number of **(** is not equal to the number of **)**, the sequence also will not be valid.

Now, we have found our solution, rest of the things are just easy implementation of segment tree with lazy propagation, update in range with +1, -1 and find min,max.

No query function needed, the root node can say if the sequence is valid or not and what is the depth.

Complexity:
Every update will take a time complexity of **log(|sequence|)** and query is answered at a complexity of **O(1)**. So, overall complexity is **Q*log(|sequence|),** where Q is the number of queries.

Other notes: Some asked during the contest whether the data was accurate. It was indeed correct, but some were getting “Output Limit Exceeded” due to outputting too many “We are doomed!” strings, i.e. the solution was incorrect. If the Output Limit had been increased, those solutions would’ve still gotten “Wrong Answer”.

Problem B (Trends):

Setter: Mir Wasi Ahmed
Alternate Writer: Raihat Zaman Niloy

Solution:

Given an array: ? ? a ? b ? ? ? c ? ?, where ? are unknown and the values of a, b, c etc are given. Find the number of non-decreasing arrays where each element may have the value 1 to **n**. We could solve each contiguous ??? independently. That is (??)a(?)b(???)c(??), each of the (...) part independently and then take their product to get the final answer. How to solve each of the contiguous ?? part? First of all we can insert 1 at the beginning of the given array and **n** at the end. Now every contiguous ?? part is preceded and tailed by some number. Say the preceding number is **i** and tailing number is **j**. That means, all these numbers has to be between **i** and **j** and non-decreasing. Suppose we have **m** ? between **i** and **j**. How many ways to assign value to **m** ?s. Consider a 2D grid. We are at the lower left point and we have a variable **v**. Initially **v** = **i**. If we go up, that means, **v** would be increased by 1. If we go right that means the value of next ? is **v**. We know our max value can be **j**. So we can go **j-i** steps up (**U = j-i**). Also we need to print m values, so we need to go m values right (**R = m**). So the problem is: How many ways to go **U** steps up and **R** steps right, which is **ncr(R+U, R)**.

Problem C (Residential Area):

Setter: Md. Samiul Islam
Alternate Writer: Hasnain Heickal Jami

Solution:

Easy problem. Since in the area there will be exactly one of the service from the ten services, so the area size would be 10. This can be in four shapes: 1x10, 10x1, 2x5, 5x2. So fix a corner (say upper left corner), you

will need 2 for loops. Then for each of this pattern, check whether there is any duplicate inside the rectangle.

Problem D (Connecting To One):

Setter: Muhammad Ridowan
Alternate Writer: Anindya Das

Solution:

One way to solve this problem is doing in reverse. Take edges from largest to smallest. Also you need to consider disjoint set (union find) data structure. For each edge, join these two nodes and record that “for this edge cost, there were this many nodes connected to 1”. Once this process is done with all the edges, you can easily answer the queries. Suppose you are asked for cost **C**. See if **C** is among edge costs, if so you have your answer from there. If not, look for the edge cost that is larger than **C** and there is your answer. I hope you would be able to figure out how to handle the case where there are multiple edges with the same edge cost. This is like, kruskal’s MST (Minimum Spanning Tree) algorithm. Another way is to do like prim’s MST algorithm.

Problem E (Anti Hash):

Setter: Sabit Anwar Zahin
Alternate Writer: Kaysar Abduallh Shoikot

Solution:

One naïve way to solve the problem is to keep generating a random string until the hash is equal to the hash of the input string (let’s call it **h)**. We can generate a random string of length **n** at first. To get the subsequent random strings, instead of generating them all over again, we can select a random index from the last string we generated, and replace its value with a random character. In this way, we can generate random strings and their corresponding hash values in **O(1).** However this is too slow, because the expected runtime of this algorithm is **O(modulus).** (Assuming all hash

values are equally likely and this will be the case since **the base and the modulus are co-primes**).

To optimize this algorithm, we can use the birthday paradox. Let's say we generate **100000** random strings of length a and **100000** random strings of length **b**, and **(a + b) = n**. Lets store these strings in set **A** and set **B** respectively. Say we want to find the string **T** such that $\mathbf{T = A_i + B_j}$ (+ denotes concatenation here). We can loop through all the strings of **A**, and for a fixed string $\mathbf{S_a}$ with hash $\mathbf{H_a}$, we need to find a string $\mathbf{S_b}$ with hash $\mathbf{H_b}$ from set B such that, $\mathbf{(H_a * base^b + H_b) \% modulus == h.}$ We can pre-process all the strings in set **B** and store their corresponding hash in a hashmap/map. So now for a fixed string $\mathbf{S_a}$ from **A,** we can check in **O(1)/O(log 100000)** whether such a hash exists in set **B.**

Assuming when the modulus is 2^31-1 (the worst-case), The probability of this algorithm successfully finding a solution is greater than 0.9. In case this fails to find a solution, we can of course run the same algorithm again and again until it finds a solution. The expected number of strings generated will be **sqrt(modulus)** (same reasoning as in birthday paradox).

Overall complexity: O(sqrt(modulus)* O(1)/log(sqrt(modulus))) * t

Problem F (Coldplay):

Setter: Hasnain Heickal Jami
Alternate Writer: Md. Samiul Islam

Solution: The easiest problem of the set. Just write the formula.

Problem G (Pattern Lock):

Setter: Md. Ashraful Islam Shovon
Alternate Writer: Aninda Majumder

Solution:
Just imagine how would you solve it. First all the seals are intact. So all of the slots are connected. That is currently **1...n** are connected. You will

apply some rotation. And then break between **i** and **i+1**. Then independently solve **1...i** and **i+1...n**. But now **1...i** and **i+1...n** are not in initial state, since you have already rotated by some amount. So the state is: what is the current range of the intact slots, and what is the amount they are already rotated. Please note, rotation of 10 means, 0 rotation. So this rotation amount can be kept modulo 10. So the state size is: 10**n**^2 (start of intact slot **x** end of intact slot **x** amount of rotation%10). Now, in every state you can rotate it in 19 ways (1...9 step clockwise or 1...9 step counter clockwise or no rotation, actually it can be optimized to only 10 ways). For each of these rotations, you would like to break the next seal (a for loop of size n). So in 10**n**^2 state you would need 19n loop inside. Base case is when the intact slot size is 1.

Problem H (Maximum Subpath):

Setter: Raihat Zaman Niloy
Alternate Writer: Md. Nafis Sadique

Solution: Key idea is Centroid decomposition.

Suppose **x** is the centroid of the current tree. Now run a DFS (Depth First Search) from **x**. For each node **y**, compute two values: $\mathbf{h_y}$ hash of all the node colors of the path **(x, y)** and the $\mathbf{e_y}$ maximum edge cost of the path **(x, y).** Now for each $\mathbf{h_y}$ you know what has to be the hash value of the remaining color nodes. Look for such hash value and find the answer. After solving for **x**, erase it, and recursively solve for all subtrees.
Let **(V1, V2)** is a pair that contains all the colors of **M** exactly once. Now, if **U** is a centroid and path **(V1, V2)** goes through **U**, then the maximum subpath value can be calculated using the following equation:

```
max_sum_of_path (V1, V2) = max( max_sum_of_path(U, V1), max_sum_of_path(U,
V2), max_prefix(U, V1)+max_prefix(U, V2) )
```

Problem I (Repeated Digit Sum):

Setter: Jane Alam Jan
Alternate Writer: Hasnain Heickal Jami

Solution:
This is a pretty standard problem. The answer is almost **f(a^b) % 9**. I said almost because, if you get 0 after modulo 9, then answer is 9 (**a**=0 is **a** special case). Why?

First of all, what's going on here? Initially **x = a^b**, then we keep summing digits of **x**, until we get 1 digit number. Suppose **x** is a **n** digit number = $\mathbf{x_{n-1}...x_1x_0} = \mathbf{10^{n-1}x_{n-1} + \ldots + 10x_1 + x_0}$. If you take modulo 9, it will be = $\mathbf{(x_{n-1} + \ldots + x_1 + x_0)\%9}$. It means, **x % 9** = sum of digits of **x % 9**. And if you keep taking mod 9, you will eventually reach one of the numbers 0..8.

Problem J (Rotation Motion):

Setter: Md Mahbubul Hasan
Alternate Writer: Muhammad Ridowan

Solution:
This is an easy geometry problem. One thing you should notice that, initial position of **A** may be left of the circle. Otherwise this is a pretty straightforward problem. From the initial position of **B**, find out the angle **B** creates with **x**-axis. Then add **t*w** with the angle to get the final position of **B**. So you know, **B**, distance **AB** and **A**'s **y** coordinate is 0. So you can find **A**'s **x** coordinate easily from the distance formula. Please note, since this is a quadratic equation, there would be two solutions, so you need to select the appropriate one (negative one if initial **A** was in negative side).