**Analysis of SUB IUPC 2019**

**Fidelity, SiamH, \_Ash\_\_, sgtlaugh**

### A. A Permutation with Mr. Peabody and Sherman

**Math.** For an index i, there are numbers that you can write down at i and get a displacement. And this displacement will belong in permutations.

So you have total displacements =

As there are toal permutations possible, average number of displacement is .

### B. Satisfy the Constraints

First think of the brute force solution. What value should we place at index **i** ? We should take every **X** corresponding to range that contains **i** and **Ai** is just LCM of all those numbers. If we think of a specific prime, then power of that prime in **Ai** is just maximum of powers of all **X** in consideration. But this is too slow. O(n^2)

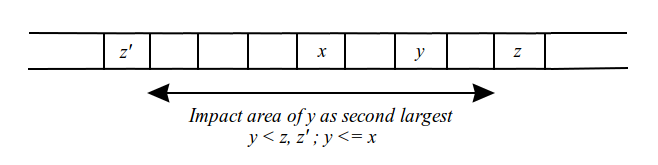
Let’s solve for each index one by one. When we are solving for index **i ,** we keep for every prime the powers that have been found for every range in consideration and current LCM considering all these prime powers. Now what happens when we go to the next index. Some ranges have started whose left point is at index **i + 1 .** We insert those range. To insert a range, just find the powers of prime , and insert their power for respective prime. Now notice that LCM may change due to this insertion. How does this change ? If the maximum of powers of that prime increases due to this insertion , then we have to multiply current LCM by Prime^(increase in power). Erase of a range may be done in similar fashion. Erase and insertion can be done efficiently using **set.** You may also have to precompute the factorization and modular inverse of all numbers upto 1000000. Each range is inserted and deleted once. And each number upto 1000000 consists of at most 7 primes. Hence complexity for each test case after precomputation : O(Q \* 7 \* log(Q) )

### C. Bittersweets of Our Community

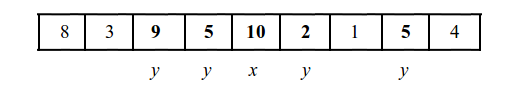
**Giveaway.** Just check whether there is a contest on that date. If there is one, check if the cities match. And if they do, check whether it’s a dinner invitation.

### D. Smartest Subarray

**Data Structures. Binary Search.** Consider any element of the array to be the largest. Suppose it’s ***x.*** Now find the second largest number for this **x,** suppose it’s **y.** So **y** <= **x.** Find the impact area of **y** as second largest; the area should be a subarray where **x** is maximum and **y** is second maximum.



After fixing **x** and **y,** finding the longest subarray should be easy and can be done in logarithmic time. Fixing **x** has a linear time and for every **x,** we’re iterating over every element as **y** so far. That turns out to be **O(*n2 lg n*)** and certainly won’t pass the time limit. One observation here is that for a **x** you don’t have that much **y** candidates. And this reduces the complexity to **O(*n lg n*)**.



### E. Carry My Stuff

**Greedy.** Create an adjacency list of products based on their types. Sort every row of the adjacency list by order time. Sort Travellers by order time. Iterate over adjacency list (by type of product). Use binary search to find the traveller with lowest arrival time who arrives on the day or after that order and has not yet been assigned a object of that type. [If binary search fails for a product, the answer is -1]

### F. Cut the Rope

**Treap on ranges, Math.** There are two parts of this problem. Let’s simplify the problem a bit and solve the first part. What if we were told to delete all the numbers in a range and asked to find the sum (instead of xor) of all the numbers between the L’th and R’th index? The idea is to have a range at each node of the treap. So we start with one node, containing the range[1:N]. To delete all the numbers from [L:R], we first cut the treap in appropriate positions. So when there is one node containing [1:N], first we divide the treap into [1:L-1], [L:R] and [R+1, N]. Now find the node containing [L:R] and delete it, and merge its left and right subtree using standard treap merge. To calculate the value (sum for the simplified version) of a node, we can use any standard formulas/methods.

To handle the periodic delete, the idea is to keep in each node of the treap an arithmetic progression instead of a range. It’s quite similar to that of keeping a range, but the split and merge can be a bit tricky to implement efficiently.

Coming back to the second part, if we need to find the xor now, the problem boils down to finding the xor of all the numbers in an arithmetic progression. It’s not very hard and can be solved in O(log^2 N), which in this case would be O(64^2). It requires a bit of math and observation, and can be implemented using simple recursion.

Overall complexity: **O(Q \* log^2 N)**

### G. Precision Error

**String matching.** If any of those numbers are missing decimal point, add at the end. Pad those two numbers with enough zeroes so that their lengths are equal and longer than k. Now just check whether the numbers have decimal point at the same position. If they are, just take the first k digits of those numbers except the decimal point and check whether they’re same.

### H. Horrible Pass

**DP. Probabilities. Combinatorics. Bitmasks.** Generate a mask *goal* which will have set bits at i’th position if i’th vertex needs to be visited to complete the mission. Now we propose a dp solution with (current vertex, vertices needs to be visited yet) states:

where is the probability to go to v from current state and .

To calculate , notice that Boko will behave optimally to complete his mission with maximal probability. Therefore, among a combination of chosen v’s to go to, he’ll go for the v that has the maximum . So, if *u* has *n* neighbours, the *v* with maximum dp value would be the maximum in combinations. The second max would be the max in combinations, the third best would be max in in combinations and so on.

So, if we sort the possible v’s according to their dp values in non-increasing order, for the i’th *v*, = . Notice that if , Boko will definitely go to the *v* with max value. Complexity is **O(*n2 2n lg n*)**.

### I. Fencing the T-Rex

The problem basically asks to compute the convex hull of the given points and find the radius of the largest circle that you can fit in the convex hull. This is a classical problem of half-plane intersection.   
 First observation is that you can binary search on the radius of the circle. Now you want to check whether you can fit a circle of radius **R** inside the convex hull. Where can you place the center of this circle ? Think of a side of the hull, the center must be at least as far as **R** from this side. That means you can shift the side (actually the line) **R** distance inside the hull and the center must be on other side of the line. Like this every side imposes this kind of constraints : **the center must be on a half-plane created by a line.** Now if there is a point satisfying all these constraints, then you can fit the circle inside the convex hull. Now how to check whether there is a point satisfying all these constraints. Here comes half-plane intersection comes into play. There are several algorithms. You may refer to this link for details : **https://codeforces.com/blog/entry/61710**

### J. Evil Corps

The problem is just polynomial interpolation and can be solved with a number of numerical algorithms. I am describing Lagrange Interpolation :   
Given we can construct the polynomial

You can compute in linear time for all by computing and so, the complexity required for each query will be linear.

[In this case you would need to handle the case of x-x\_i for some i separately to avoid zero division error.]