BUET INTER UNIVERSITY PROGRAMMING CONTEST 2023

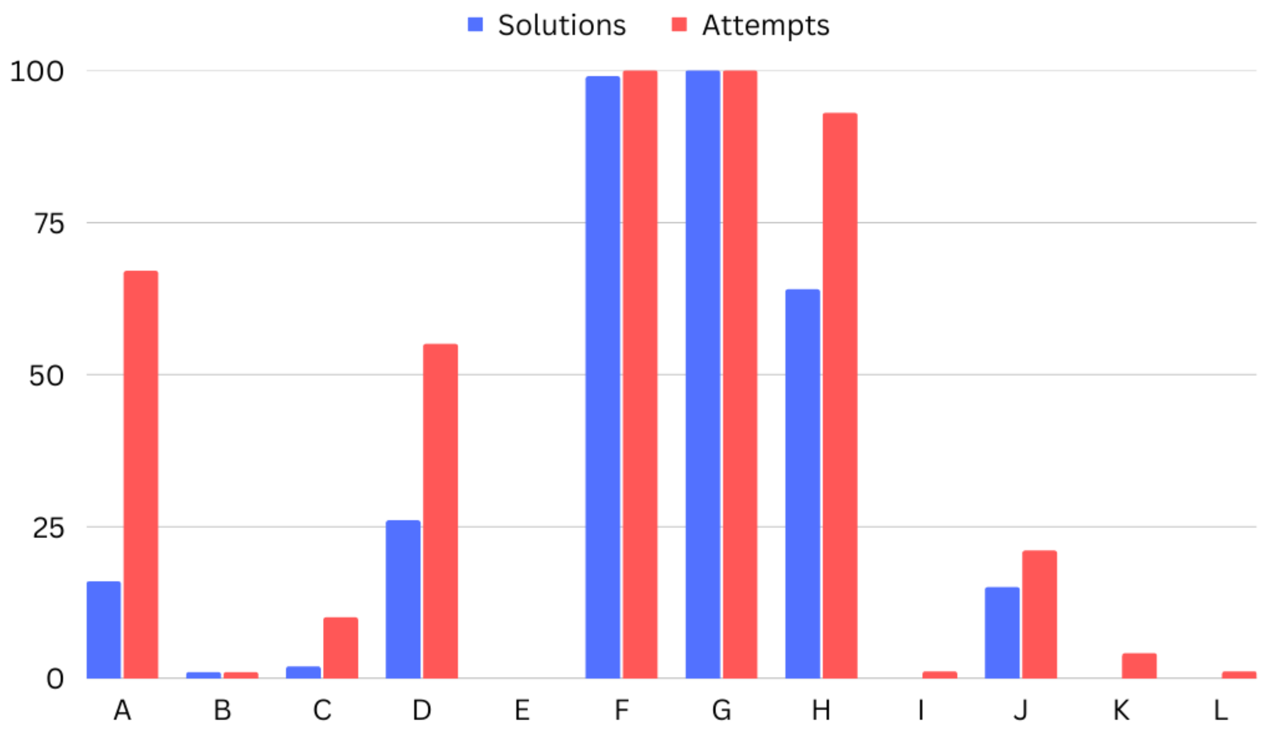
Problem Set Analysis

Table Of Contents

A. Satisfaction....................2
B. Peculiar Partitioning II.................... 3
C. The Very Last Exam....................5
D. Festive Shuffling....................6
E. Ankara Messi....................7
F. Chocolate....................8
G. LR.................... 8
H. Better Together!....................9
I. Reassemble....................10
J. Euler's Peculiar Dream.................... 12
K. Cold Rainy Night in Stoke.................... 14
L. Gardening....................15

Statistics

Total Problems: 12
Total Teams: 102



A. Satisfaction

Setter: Arghya Pal
Solve count: 16

1. Let the total number of contests be C. It is optimal to use the first C problems.
2. We will use assign problems to contests in increasing order of interesting factor i.e from C^{th} to 1^{st}.
3. We will assign the i^{th} problem to the least important unassigned contest possible. We can maintain the list of unassigned contests using various data structures like priority queue or set.

B. Peculiar Partitioning II

Setter: Pritom Kundu
Solve count: 1

Let $f(x)$ be number of partitionings with score x. Let $g(x)$ be the number of partitionings with a score which is a supermask of x. Instead of calculating f, we will calculate g. Then f can be calculated from g with inverse SOS dp.

Let's look at how to calculate $g(x)$. The score of a valid partitioning must be a supermask of x. If the ith bit of x is 0, the score's i'th bit can be anything. So, we don't have to worry about these bits. If the ith bit of x is 1, the score's ith bit must be 1. Thus one of the two partitions must have all numbers with i'th bit 1. Alternatively, we can say that all numbers with i'th bit 0, must be on the same partition.

So, to calculate $g(x)$, we need to count ways to partition such that for each on bit in x, all numbers with that bit 0, is on the same partition. This allows a solution with DSU on the array indices, For each on bit, we merge the indices which have that bit 0. Then, $g(x)$ is simply 2^{c} where c is the number of components. This solution is unfortunately too slow.

To speed it up, instead of keeping a DSU over indices, we will keep one over bits. Two bits i and j will be merged, if both of them are on in x and there is at least one value with both bits off. Because in that case, every number with at least one of these bits off, will be in the same partition.

After merging for all on bits, consider the components formed. Each component consists of some bits, and all numbers with any of these bits 0 must be on the same partition. So, each component contributes a factor of 2. We will also need how many numbers will be covered by this component. Let $mask$ be the bitmask of bits in the component. Then, the component will cover the numbers with at least one of the bits in mask 0, this is equal to $n - h(mask)$ where $h(x)$ is the count of numbers which are supermasks of x. h can be easily calculated using SOS dp.

However, there will be some numbers which won't be included in any of the components. These are independent numbers, and can be placed in either partition. Suppose there are $p$of them. Thus $g(x) = 2^{c} \cdot 2^{p}$. We calculate g for each possible x. Then we can get f with an inverse SOS dp.

Overall complexity is $O(k^2 2^k)$

C. The Very Last Exam

Setter: Sabbir Rahman Abir
Solve count: 2

Since the product of $k + 1$ for all bags is at most 10^{12} which is also all possible way to choose a set of stones, this implies trying a meet-in-the-middle approach. But we cannot simply divide the bags in equal parts, rather we have to ensure that the two parts are close in terms of the products. If we keep taking bags and stop when product of $k + 1$ crosses 10^5 then it is also confirmed to be less than or equal to 10^7. So creating such two parts will ensure that the two parts have between 10^5 to 10^7ways to pick stones.

After dividing into two parts, the next difficulty is what to compute and how to merge the two parts so that the average is exactly p/q. This may seem difficult as for both parts we'll have to track the sum of picked stones and count of stones picked to get average weight. But instead of keeping track of both sums and counts, for a particular way to pick stones with sum and $count$ we can save $q \cdot sum - p \cdot count$. From the first part we get a list of such expressions and we also get a list of such expressions from the second part. If two expression $q \cdot sum_1 - p \cdot count_1$ and $q \cdot sum_2 - p \cdot count_2$ sum to 0 then these two parts combined have target average as
$q \cdot (sum_1 + sum_2) - p \cdot (count_1 + count_2) = 0 \Rightarrow \frac{sum_1 + sum_2}{count_1 + count_2} = \frac{p}{q}$.

So the final solution is to create the two partitions and get two lists of expressions. And then count pairs from two list that sum to be 0. There are several things to optimize, such as using recursion to store expressions from the first part and then using recursion on the second part to only generate and binary search for the expressions in the first part's list. There was also a corner case of reducing the target p/q as without doing so results in overflow and wrong answers.

D. Festive Shuffling

Setter: Sachin Deb
Solve count: 26

You need to use 3 state dynamic programming here. First two states are l and r which mean the first and the last position of the current subsegment, and the third state is the number of 1's in between the special positions which are situated in [l, r] range.

The transitions are like this:

1. If both of l and r are special positions then you can check if you can put 1 in both positions or 0 in both positions. In both cases the ans will increase by 2. If you can't then the answer will be the same as the answer for (l + 1, r) or (l, r - 1) according to what we have put on l and r.

2. If only l is a special position then we need to check if we can make l equals to r by shuffling or not. You will change the answer according to your assignment to index l. The same is true if only r is a special position.

3. If none of them are special position then you can do what you would do in normal Longest palindromic subsequence problem, maintaining 3 transitions:
    a. if already index l and r have the same element. Transition to (l + 1, r - 1)
    b. if they don't have the same element. Transition to (l, r - 1) or (l + 1, r)

Carefully compute the value of the third state. See solution code for more details.

Overall complexity O(n^3)

E. Ankara Messi

Setter: Kazi Md. Irshad
Solve count: 0

We will be referring to Messi as ‘player’ and Gvardiol as ‘circle’.

Let us visualize a non boring dribble first. Let us assume the player leaves contact with the circle, but still cannot directly run towards the origin without getting intercepted. Let such a locus which reaches the origin eventually be L. The player can incrementally make L a little bit shorter until the circle touches the player. Therefore a locus where the player after leaving contact of the circle, cannot directly run towards the origin without getting intercepted is boring. Which means a non-boring dribble is the locus where the player remains in contact with the circle for some time, then runs directly towards the origin.

Starting to overtake the circle from one direction then switching to do it from the other direction does not help. There would come a time when the player, circle-center, and origin are collinear, which can be achieved even faster by moving away from the circle at speed 1 from the start.

Let us try to find the shortest non-boring dribble. From the first moment when the origin is reachable by directly running towards it, it becomes suboptimal to stay in touch with the circle any longer. Because any other path cannot be shorter than the straight line. By this reasoning, the angle between origin, player and circle-center is constant for the moment the player starts running straight. The player can maintain a maximum angular speed of sqrt(v*v-1) with the circle. Going with a lower angular velocity at some point in the locus is equivalent to moving away from the origin at the start at speed 1 for some time then moving with this maximum angular speed. This is suboptimal because an arc of a circle is the shortest path around that circle.

Now let us find the locus. The player has angular speed of sqrt(v*v-1). From this we can find the velocity vector for the circle-center with respect to time. Then we can find the displacement vector of the circle-center with respect to time. We can find the displacement vector of the player by adding the above mentioned displacement vector and velocity vector. You can find the final calculated locus in this link. The green circle represents the player locus. And the red line represents the straight run towards the origin.

F. Chocolate

Setter: Sachin Deb
Solve count: 99

In this problem there might be an illusion that I may buy x of first type chocolates and then n - x of second type chocolates. But in reality you only need to buy a certain type of chocolate and this will be the optimal situation. Why is it true?

Say you have bought x first type of chocolates, and the price has decreased by (x - 1) * d1. Now if you try to buy the second type of chocolates for the rest then you can clearly understand it would never leave you in a profitable solution.

Because if buying the second type of chocolates is profitable then why not buy it from the beginning? and the same goes for the reverse situation, if buying the second type of chocolates are not profitable then why should we leave the first type of chocolates and move to the second one. From the above two conditions it is clear that it is better to stick to only one type of chocolates.

Complexity for answering each query: O(1)

G. LR

Setter: Ahmed Hossain
Solve count: 100

The answer is $r\ /\ 2\ -\ (l\ -\ 1)\ /\ 2$

H. Better Together!

Setter: Mehbubul Hasan AL Quvi
Solve count: 64

G_t = the graph in which for each pair of k = 1, 2, … , t and x = 0, 1, 2, … , N - 1, there is an edge (road) of cost C_k between vertex (city) x and vertex (city) $(x + A_k)$ mod N. The problem is then equivalent to that of finding the MST in the graph. But, in this problem the graph has NM edges, so applying plain Kruskal's algorithm will be slow.

First, we sort C_k in the ascending order. The application for G_M will be:

1. First, we add edges to T as much as addable with road-construction of type 1.
2. Then we add to T as much as addable with road-construction of type 2.
3. And all the road-constructions till road-construction of type M are repeated similarly.

Let X_t be the number of connected components of G_t. On G_t, for any k = 1, 2, … , t and any x = 0, 1, … , N - 1, we can move from vertex x to vertex $(x + A_k)$ mod N (or vice versa) along an edge, so on G_t, one can travel from v to w iff "there exist integers k_1, k_2, … and k_t such that $w = v + k_1A_1 + k_2A_2 + … + k_tA_t$ (mod N)". This condition is equivalent to: $w \equiv v \pmod{d_t}$; [$d_t = \gcd(N, A_1, A_2, …, A_t)$]. So, vertex v and vertex w belong to the same connected component iff $w \equiv v \pmod{d_t}$, so $X_t = d_t$.

The number of edges added by the step of "adding as many edges as possible with operation t" is $X_{t-1} - X_t$. ($X_0 = N$). Therefore the answer for the problem is $\mathrm{Sum}(C_t (X_{t-1} - X_t))$ from t = 1 to t = M. If $X_M > 1$, then G_M is not connected, so the answer will be -1.

I. Reassemble

Setter: Tariq Bin Salam
Solve count: 0

Let, f_i be the minimum distance needed to cross by troop i.

$f_i = i \cdot |d_i - d_j|$ where $j = p_i$, the post where troop i needs to be stationed after all commands are applied. Remember, troop i is stationed at post i at the beginning.

The minimum distance needed to cross by all soldiers, $D = \sum_{i=1}^{N} f_i$

It can be proved that all troops can be stationed at their desired post by crossing a total of D distance only. The proof goes as follows.
We can always find a pair of posts, i and j such that $i > j$ and $p_i \leq j$and $p_j \geq i$ until the shuffle is complete, because each post i $(1 \leq i \leq N)$ was filled at the beginning and will be filled after the shuffle. So the troop currently stationed at post i wants to go to the left and the troop currently stationed at post j wants to go to the right. If we swap post i and j, no troop will cross more distance than they need to. So we can keep swapping like this until all troops are at their desired position. This way the troop i will cross exactly f_i distance and the total distance crossed by all soldiers will be D.
Now to guarantee that we have done a minimum number of swaps, we need to pick i and j such that they belong to the same cycle. Note that, if we swap two posts i and j in the same cycle, the cycle will break in two cycles (or two loops or one cycle and one loop). So each move will increase the number of cycles by 1 until we have N loops (consider the loop as a cycle of length 1). We cannot do less swaps than this, because any swap cannot increase the number of cycles by more than 1.
To find a pair that follows above conditions, we can first take a cycle, then find the minimum i (the leftmost post) such that $p_i < i$ in that cycle. Now for any post $j \in [p_i, i)$ that also belongs to the same cycle as i we have $p_j > j$. So we can take the maximum $j \in [p_i, i)$ and that will lead to $p_j \geq i$ otherwise the cycle cannot be completed. So we can swap i and j. Also remember to swap p_j and p_i . So after this swap we have $p_j \leq j$ and $p_i \geq i$. If $p_j < j$, that means j is the leftmost post such that $p_j < j$ and we can do the above again. Otherwise $p_j = j$

and we can move to the second minimum post i such that $p_i < i$ in the cycle and keep doing it until the cycle is done. **All this can be done in** $O(n)$ **or** $O(n \cdot logn)$.

J. Euler's Peculiar Dream

Setter: Hasanul Islam, Ahmed Hossain, Sk Sabit
Solve count: 15

In this text all p or $p_{<subscript>}$ denotes prime.

Let $x = \prod_{p_i | x} p_i^{e_i}$ --(eq1) be the canonical factorization of x which is given in the input.

Now we know $\phi(x) = \phi(\prod_{p_i|x} p_i^{e_i}) = \prod_{p_i|x} p_i^{e_i - 1} * (p_i - 1)$

Let's make a directed graph G, where each prime p $(2 \le p \le 10^6$ corresponds to a node in G. This graph has about $k = 78000$ nodes and $O(k\, log(10^6)$ edge. Now let,

$p_i - 1 = \prod_{p_j | p_i - 1} p_j^{e_j}$ be the canonical factorization of $p_i - 1$

Then we ad an directed edge from p_i to p_j for all j with weight e_j.Now this is a DAG(Directed Acyclic Graph) since an edge between between $p_1 -> p_2$ means $p_1 > p_2$ so there can't be any cycle. Also note this DAG has only one sink that is 2.
We can model applying the totient function on a number as an operation on this graph.
Initially we place a value e_i (from eq1) to each node p_i and every other node has a value 0. Then each application of totient function reduces value on each p_i by 1 and increases the value of of all neighbors p_j by e_j simultaneously.
We will reach at 1 when every node on G has a value of 0. But since 2 is the only sink, the sum of value of all nodes can decrease only at 2 by 1 each operation. **Note that the value of 2 can be 0 only at the beginning and after all other node's value has been reduced to 0.** Since 2 is a factor of all $p - 1\ if\ p > 2$
So we compute the contribution of a node to the value of 2 by dp. Since this is a DAG we can do it in $O(node + edge) = O(max\{p_i\} * log(max\{p_i\})$ time complexity.
We use the following dp formulation:

$$dp[p_i] = \sum_{p_j} e_j * dp[p_j]$$

Here $dp[p_i]$ denotes if we place value 1 at p_i and 0 to every other value then apply totient function repeatedly on this Graph then the value of 2 will increase by $dp[p_i]$ overall.

Then the $answer\ = (\sum_{p_i} e_i\ *\ dp[p_i]) +\ (1\ if\ 2\ isn't\ a\ factor\ of\ x)$

Overall time complexity is $O(max\{p_i\}\ *\ log(max\{pi\})$

K. Cold Rainy Night in Stoke

Setter: Iftekhar Hakim Kaowsar
Solve count: 0

Short statement:
Given *N* points and a circle. Construct a convex polygon using a subset of the points so that it encloses the circle completely.

Solution Idea:
First create a list of **directed** segments using all pair of points. We can ignore segments with zero length. We will only care about a directed segment *AB* if it keeps the circle on the left and does not intersect the circle when extended infinitely. Discard others from the list. We can surely tell that if we create a polygon with these directed segments only, it will surely enclose the circle. But it may not be optimal, if we consider the area. And it may not be convex too.

Now consider each directed segment *AB* as a vector *v=B-A.* Do a polar-sort on the list of all good segments using their corresponding vector. Now we are sure that if we create a polygon using a subsequence from the list, it will be convex. And we can create any possible convex polygon using some subsequence. So, we are left with choosing the optimal convex polygon.

Let the cost of directed segment *AB* be the area of triangle *OAB.* Consider *dp[i][j]=minimum area of possible convex poly-line which starts at i and ends at j.* We select a directed segment *AB* serially from the sorted list and try to merge point *B* with *dp[i][A]*. So, dp[i][B]=min(dp[i][B],dp[i][A]+cost(AB)), for all *i*. Hence we can compute the *dp* table using dynamic programming in *O(N^3).* Finally we will find the minimum value of all *dp[i][i]* to get the actual answer.

L. Gardening

Setter: Sk. Sabit Bin Mosaddek
Solve count: 0

Let’s break down each query in two parts. The first part is the sum of the distance of all vertices from vertex R without any modification and the second part is calculating the improvement due to adding the edge U - V.

We can precalculate the first part for all possible values of R. We set a root (let’s say 1). Then we calculate the sum of distances from this root and calculate the subtree size of each node (let’s say Sub_i). After that we can calculate the sum of distances for each node using the answer for its parent. If the sum of distances from vertex X is Sum_X, then

$$Sum_X = Sum_{par[x]} + n - 2 * Sub_x$$

Using this formula we can recursively calculate Sum_x for all possible values of X.

Now, we have to improve our answer using the given edge. We can use LCA and the idea of binary lifting to find our answer. Let’s say the LCA of U and V is L when the root is 1. Now, we have to find the LCA when the root is R, let’s say this LCA is L’. There are 3 possibilities. Either L = L’, or L’ is on the path from L to U, or L’ is on the path from L to V. We can find the LCA of (R, U) and (R, V) then check if they are ancestors of L or not. If both are ancestors then L = L’ otherwise whichever is not the ancestor is the new L’.

Now we have L and L’. We can calculate the distance from L’ to U and L’ to V. Say, dis(L’, U) > dis(L’, V), then after adding the edge U - V, it is only possible to improve some distances of vertices which lie on the subtree of some vertices which are situated on the path from U to L’. The improvement will happen up to the K'th vertex along the path front U to L’. Here, K = (dis(L’, U) - dis(L’, V)) / 2. Let’s say the vertices on the path from U to L’ are U, U_2, U_3 , …, U_K, …, L’. Now, we have two cases. i) U_K comes before L. ii) U_K comes after L.

We will solve these two cases separately. In the first case and when dis(L’, U) - dis(L’, V) is odd, the change in answer is like this –

Change = $-2*(Sub[U_K]-Sub[U_{K-1}]) - 4*(Sub[U_{K-1}]-Sub[U_{K-2}]) - 6*(Sub[U_{K-2}]-Sub[U_{K-3}])$ -..

If we reduce the sum, we get, Change = -2 * (Sub[U_K]+ Sub[U_{K-1}] + Sub[U_{K-2}] + … + Sub[U]).

We can calculate this sum by binary lifting. When dis(L’, U) - dis(L’, V) is even, the equation is almost similar except that the values are multiplied by 1, 3, 5, 7, … we can calculate it similarly.
For the second case we have to handle the two parts separately, from U to L and from L to U_K. The calculation is similar to the previous case with some extra case analysis.

Binary lifting works in O(logN). Thus-
Overall Time complexity : O(NlogN + QlogN)