AGM Qualifying round solutions

AGM committee

February 2020

Task Emojis

Author:	Alexandru-Nicolae Postu
Attempted by:	52 teams
Solved by:	34 teams
Time to first solve:	29 minutes

In this task, you are given an array of N numbers and are asked to print the length of the longest non-decreasing subsequence. Moreover, the number of numbers removed that you are asked to print is, in other words, the minimum number of values placed between the elements of a subsequence of said length.

Main observation:

This problem is a variation of the longest non-decreasing subsequence algorithm. The question that arises is how we can implement it while making sure that the longest subsequence chosen also has the minimum number of numbers removed. It turns out that there are many variants. Below, one of them will be explained.

Segment tree approach:

Let $dp[i]$ be an array storing the longest non-decreasing subsequence found up until index i and the position of its first element in the main array. When updating $dp[i]$, we first check for the maximum length. Then, in case of equal lengths, we pick the subsequence with the rightmost first element. This guarantees that the number of numbers removed will be minimised.

Let us have a segment tree storing the rightmost position at which a certain value is found. To update this data structure, we use the same approach as when updating $dp[i]$: we first pick the position with the subsequence of greater length; then, we pick the one with the rightmost starting position.

Now, let us iterate through the array of numbers, $val[i]$. At position i, we will get the maximum value stored in the segment tree which belongs to the interval $[1, val[i]]$. If a value is found, then the length stored by $dp[i]$ will increase by one and the start position will stay the same. Otherwise, the length stored by $dp[i]$ will be equal to 1 and the rightmost start position, to i. That is to say, either a current subsequence's length will increase by 1, or a new one will start at position i. Moreover, for every i we update the segment tree with i and $val[i]$.

Finally, after iterating through the array of numbers, we will simply look for the longest non-decreasing subsequence. In case of equal lengths, we will pick the one with the rightmost starting position.

The correctness of the solution is insured by the fact that if we were to choose between a length x with 0 numbers removed and a length $x + 1$ with an infinite number of numbers removed, then we will choose the second option due to the fact that we want to prioritize the length. In other words, the pairs of possible answers have a total order, allowing us to use dynamic programming on them even if they are pairs.

The final time complexity is $O(N * log(N))$.

Task Magic Wand

Author:	Luca Seritan
Attempted by:	55 teams
Solved by:	33 teams
Time to first solve:	21 minutes

In this task, you are given an array of N integers and you are asked to minimize the cost of sorting it. The only operation you can do is sort a subsequence of arbitrary length k with cost k^3.

The main observation here is that we should only ever worry about sorting subsequences of length 2. It is easy to see that the cost of doing so is 8 * the number of inversions in the array.

Since in an array of size k there are at most $\frac{k(k-1)}{2}$ inversions and $8 * \frac{k(k-1)}{2} \leq k^3$ for any $k \geq 2$, we can conclude sorting only subsequences of length 2 is indeed optimal. Therefore, the answer to the problem is 8 * the number of inversions in the initial array.

To compute the number of inversions, one can use merge sort or an efficient data structure such as a binary indexed tree or a segment tree.

The final time complexity is $O(N * log(N))$.

Task Beggars

Author:	Teo Ionescu
Attempted by:	41 teams
Solved by:	19 teams
Time to first solve:	22 minutes

In this task your are given the an interval of time $[0, d]$ and n other intervals of time $[x_i, y_i]$, such that $0 \leq x_i < y_i \leq d$. You are asked to find the maximum number subsets of indexes i ($1 \leq i \leq n$) such that:

1. Each index can be found in *AT MOST* one subset
2. The intersection of the intervals of time corresponding to any two indexes in any of the chosen subsets is the *emptyset* ($[x_i, y_i) \cap [x_j, y_j] = \emptyset, (i \neq j, i \in s, j \in s, s \subseteq \{1, 2, ..., n\})$)
3. The reunion of the intervals of time in each subset is the interval of time $[0, d]$ ($\cup_{i \in s, s \subseteq \{1,2,...,n\}} [x_i, y_i] = [0, d]$).
4. Let's that the set of timestamps S, such that for any timestamp $s \in S$, then there exists $i \in \{1, 2, ..., n\}$ such that $s = x_i$ or $s = y_i$. Any timestamp $s \in S$ should be found in *AT MOST* one of the chosen subsets of intervals.

The first restrictions lead us to modeling a flow network where each of the input intervals $[x_i, y_i]$ represents an edge from the timestamp x_i to the timestamp y_i with a capacity of 1 (restriction no. 1). If we were to print out the maximum flow from node 0 to node d we would get a wrong answer, because the 4th restriction is not met. To solve this issue should limit the outgoing flow from any node (besides 0 and d) to 1. To do this, instead of connecting the node the node x_i to the node y_i, we will instead use some auxiliary nodes. We will connect x_i to the node y_i* with an edge of capacity 1. In the end we will connect all the extra nodes to their corresponding nodes with edges of capacity 1 (i.e connecting y_i* with y_i for each $i \in \{1, 2, ..., n\}$). The answer will be the maximum flow of this network.

This final complexity will be $O(d^2 * n)$.

Task Fast Race

Author:	Vlad-Andrei Munteanu
Attempted by:	22 teams
Solved by:	17 teams
Time to first solve:	126 minutes

In this task, you are given a list of races, and for each race, you know the number of participants, their initial positions on Ox axe, and their speed. Given all these and the fact that a race lasts for 10^{18} units of time, you should say, for each race, what is the final scoreboard and after what time the scoreboard does not change anymore.

Main observations:

1. Given that the positions and the speeds are all between 0 and 10^5, we can notice that, given any pair of participants, the faster one will get in front of the slower one in at most 10^5 units of time. This happens because if the faster one has the configuration $\{p1, s1\}$ and the slower one has the configuration $\{p2, s2\}$, then the amount of time needed for the first one to get in front of the second one is the maximum value between 0 and $(p2 - p1)/(s1 - s2)$. Now, using the restrictions given in the problem, we can prove that this value is at most 10^5.

2. Therefore, if we sort the participants decreasingly by their speed, and in case of equality, we sort them decreasingly by their position, then we can obtain the final scoreboard. This can be easily achieved by using counting sort for linear time or any $O(N * log(N))$ method.

3. To answer the second question, we will use the formula from the first observation, and the fact that if, at a time T, participant i is in front of participant j and participant j is in front of participant k, then participant i is also in front of participant k. This means that if, at a given time T, for all i between 1 and $n - 1$, the fact that participant i from the final scoreboard is in front of participant $i + 1$ from the final scoreboard holds, then the scoreboard will not change anymore after that time. Therefore, all we have to do is, for the final scoreboard, compute the time needed for i to get in front of $i + 1$, using the formula stated above, and chose the maximum value among all of these as your answer. This value has the property that it is the first time after which the final scoreboard is achieved, and it also has the property that after this time the scoreboard will not change anymore.

The final time complexity is $O(N * log(N))$ or $O(N)$ per race, depending on the sort method you used.

Task Government

Author:	Teo Ionescu
Attempted by:	20 teams
Solved by:	16 teams
Time to first solve:	76 minutes

In this task your are given:

- m - the number of cities ($1 \leq m \leq 30$)
- n - the number of projects ($1 \leq n \leq 30$)
- an initial m-tuple $(b_1, b_2, ...b_m)$ (the budget for each of the m cities)
- a description for each project in the form of a pair $(goodScheme_i, badScheme_i)$
- each scheme is in the form of an m-tuple $(c_1, c_2, ..., c_m)$ (the cost associated to each city for that scheme).

Your task is to choose $EXACTLY$ one of the two schemes for each project such that the the number of bad schemes is $MINIMIZED$ and the sum of the chosen schemes is $EXACTLY$ the initial budget for the m cities. Having chosen the subset $(scheme_1, scheme_2, ..., scheme_n)$ the following should take place: $\sum_{i=1}^{n} scheme_i = (b_1, b_2, ..., b_m)$.

Note that $scheme_i + scheme_j = (c_{i1}, c_{i2}, ..., c_{im}) + (c_{j1}, c_{j2}, ..., c_{jm}) = (c_{i1} + c_{j1}, c_{i2} + c_{j2}, ..., c_{im} + c_{jm})$.

Given the low number of projects and cities we can take a meet-in-the-middle approach. We will split the n projects in half. For each correct subset of projects in the first half we will calculate the cost for each city and then store the resulting m-tuple and the number of bad schemes for the respective subset (we could use a *map* or a *set* for this). For the second half we will take a similar approach, but instead of storing the m-tuple, we will check if the complement of the resulting m-tuple (relative to the initial budget for the m cities) can be found in the first half.

The answer will be *impossible* if there was no match, or the least number of bad schemes from each of the found subsets.

The final complexity will be $O(2^{n/2} * m * n)$

Task Pussycat

Author:	Teo Ionescu
Attempted by:	20 teams
Solved by:	11 teams
Time to first solve:	133 minutes

In this task your are given a DAG (Directed acyclic graph) with n nodes and each of its nodes is assigned a value. You are asked to find out if there exists a permutation $S = \{s_1, s_2, ..., s_n\}$ of the set $\{1, 2, ..., n\}$ such that:

- if node with index s_i is in the subtree of the node with index s_j, then $i < j$
- if the value assigned to then node with index $s_i = v$, then $v \geq i$

We claim that if a solution exists, we can find a good permutation by sorting the nodes in ascending order or their values and, for each node in the sorted list, if it is not marked, then mark it and its whole subtree, and recursively add all new marked nodes to the solution.

Proof: Let's assume that there exists a permutation $S = \{s_1, s_2, ..., s_n\}$ that satisfies the problem's constrains. Let's also assume that there exists a pair of nodes s_i and s_j, such that they are in different subtrees, $i < j$ and the value of s_i, v_i is greater than the value of s_j, v_j ($v_i > v_j$). Then, since we assumed that the permutation satisfies the problem's constrains, we know that $v_j \geq j \rightarrow v_i > j$, so we can swap the s_i and s_j and we still have a correct solution.

The final complexity is $O(nlog(n))$

Task The Sacred Texts

Author:	Alexandru Enache
Attempted by:	13 teams
Solved by:	8 teams
Time to first solve:	153 minutes

In this task, you are given a matrix of numbers and queries of 2 types: change the value of an element, find the maximum sum of a submatrix of a certain submatrix. You are asked to print the answers to the queries of type 2.

Main observations:

1. The maximum sum of a submatrix can be found by iterating the first and last line and then finding the maximum sum of a substring of the array formed by the sums of the elements on each column that are between the 2 lines.

2. A data structure that supports both element updates and queries of the maximum sum of a substring in a certain interval is a segment tree which stores 4 values in each node:

- *sum* - the sum of the numbers in the interval.
- *maxl* - the substring with the biggest sum that starts in the leftmost position of the interval .
- *maxr* - the substring with the biggest sum that ends in the rightmost position of the interval.
- *subst* - the substring with the biggest sum from the interval.

How to compute the 4 values of node a_x using the values of node a_{2x} and a_{2x+1}:

- $a_x.sum = a_{2x}.sum + a_{2x+1}.sum$
- $a_x.maxl = max(a_{2x}.maxl,\ a_{2x}.sum + a_{2x+1}.maxl)$
- $a_x.maxr = max(a_{2x+1}.maxr,\ a_{2x+1}.sum + a_{2x}.maxr)$
- $a_x.subst = max(a_{2x}.subst,\ a_{2x+1}.subst,\ a_{2x}.maxr + a_{2x+1}.maxl)$

3. The maximum value of N is only 10 so we have enough memory to store N^2 segment trees of length M (one segment tree for each pair of indexes i and j representing an interval of lines).

For queries of type 1 we have to update all the segment trees that contain the line of the element that is changed. The time complexity for queries of type 1 is $O(N^2 * log_2 M)$.

For queries of type 2 we have to query all the segment trees that are in the given interval. The time complexity for queries of type 2 is $O(N^2 * log_2 M)$.

The final time complexity is $O(N^2 * M + Q * N^2 * log_2 M)$.

Task KFC

Author:	Patrick-Cătălin-Alexandru Sava & Alexandru Enache
Attempted by:	18 teams
Solved by:	7 teams
Time to first solve:	47 minutes

In this task, you are given an array of N numbers and a value K and are asked to print the largest LCM of 2 elements of the array using at most K increments.

Main observations:

1. K will be distributed to at most 2 elements because there is no point in not doing so.
2. If $K > 50$ then we can make any 2 elements become co-prime. Thus resulting in a greedy algorithm in which we take the 2 largest elements and distribute K to them in such a way that their LCM (which is now their product) is maximized. In most cases this will result in 2 consecutive numbers, but there are some particular cases (mostly when K is still relatively small) where the numbers are just co-prime. This greedy can be done in many ways, some of them almost having a complexity of $O(1)$.

3. If $K \leq 50$ then there can calculate dp_{ijk} - the maximum LCM starting with the numbers i and j and using k increments. dp_{ijk} can be computed in the following way:

- $dp_{ij0} = LCM(i, j)$
- $dp_{ijk} = max(dp_{ijk-1}, dp_{i+1jk-1}, dp_{ij+1k-1})$

So the solution is created by combining the greedy solution for $K > 50$ and the dynamic programming approach for $K \leq 50$.

The final time complexity is $O(MAXVAL^2 * LIMK)$, where $MAXVAL$ represents the maximum value of the elements ($MAXVAL = 1000$) and $LIMK$ the limit for K from which we can apply the greedy approach ($LIMK = 50$).

Task BLAT

Author:	Patrick-Cătălin-Alexandru Sava & Alexandru Enache
Attempted by:	8 teams
Solved by:	4 teams
Time to first solve:	48 minutes

In this task, you are given an undirected graph with N nodes, each having a letter assigned to them and are asked to find the Kth lexicographical smallest path.

Main observation: We can find the path one letter at a time.

The first letter can be discovered by iterating it from 'a' to 'z' and counting how many nodes have this letter assigned - lets call this number X. There are N possible paths that start with one particular node, so there are $P = N * X$ paths that start with this letter. If $P < K$, then the path that we are looking for is lexicographical bigger than all of those that start with this letter, so it starts with a lexicographical bigger letter. Because we have excluded all the paths that start with this letter, the path that we are looking for is now the K-Pth smallest path from the remaining ones so we can subtract P from K. If $K \leq P$ then we have found the first letter of the path.

The remaining letters are found in a similar manner but firstly we need to calculate some values for each node:

- M_{ij} - a map where we store the weight of the subtree of node i, considering j as its parent ($1 \leq i, j \leq N$, $i! = j$, there is an edge between i and j).
- dp_{ij} - the number of paths that start with node i and continue with a node that has the letter j assigned to it.
  In order to compute it, $dp_{ij} = \sum M_{ki}$ where node k is adjacent to i and it has assigned the letter j to it.

Both M and dp can be calculated with a simple DFS (Depth-first search).

Now, to determine a letter knowing the letters before it (thus knowing the paths that have these letters) we just have to calculate $paths_l$, the number of possible paths that start with the current one and have the next letter equal to l. $paths_l$ can be calculated by adding dp_{il} for each path with the last node equal to i. But if the second last node of a path has the letter l assigned to it we have to subtract M_{ji} where j is the second last node (because a path cannot have the same node twice in it). The only thing left to do is to apply the same algorithm from before where if $paths_l < K$ then K becomes $K - paths_l$. After determining a letter, keep in mind that that each path can end right now, so if K is less or equal to the number of paths then we have found the path. If K is bigger then we subtract the number of paths from it. This algorithm continues until we have found the path.

The final time complexity is $O(Mlog_2M + Klog_2M)$.

Task PokerStars

Author:	Patrick-Cătălin-Alexandru Sava
Attempted by:	3 teams
Solved by:	None
Time to first solve:	N/A

In this task, you are given a game of poker with N players, each of them having 5 cards and are asked to print the probability of winning the game for each player.

The problem is mostly based on implementation, the real difficulty being the fact that there are many poker hands and a lot of criteria for determining the winning one. There are a lot of methods for doing this, but a separate function for each hand is recommended.

At the end there is another challenge, the output format. For each player i, you should output the probability of winning the game as $P_i * Q^{-1}$ modulo 100055128505716009 (which is prime). This is done using the fast exponentiation, modular inverse and logarithm time multiplication (in order to avoid overflow) algorithms.

The time complexity for this task shouldn't be a problem, the big time limit encouraging a cleaner approach rather than a very optimized one.

Task Security Cameras

Author:	Luca Seritan
Attempted by:	13 teams
Solved by:	None
Time to first solve:	N/A

In this task you are given a directed graph and you should find the nodes that lie on all the paths from a node A to another node B. We will call such nodes critical.

First, find any path from A to B and let this path be $x_0x_1...x_L$ such that $x_0 = A$ and $x_L = B$. Obviously, these are the only candidates we should consider in our solution. Now, we will call node x_i to be more advanced than x_j if $i > j$.

Claim: a node x_i is critical if and only if it is the most advanced node that can be reached starting from A on any path that doesn't pass through x_i itself.

This gives the idea of the following solution: we will process the nodes on the path one by one while maintaining our main candidate: the most advanced node we have been able to reach thus far.

While processing node x_i, if this is the most advanced found on any path that may pass through nodes $x_0..x_{i-1}$ then add it to the solution. Now, the only paths starting from A that haven't been considered are the ones that pass through x_i. We can simply consider them by running a BFS from x_i. In this BFS, you should not go to previously visited nodes and you should also not go to any other x_j. Instead, if you find a neighbour x_j, update the most advanced node found so far if that is the case.

The final time complexity is $O(N + M)$.

Task Unpredictable Tree

Author:	Bogdan Sitaru
Attempted by:	1 team
Solved by:	None
Time to first solve:	N/A

In this taks your are given the root of a tree and asked to perform some operations on it.

Let's consider the DFS (Depth-first search) *linearization* for our tree (we'll traverse the tree and store, in order, for each node, the first time we reach it, as well as the last time). Initially, our tree contains just the root node 0, so the *linearization* is $(0_f, 0_l)$, where the f index signifies the first entry of the node and the l index signifies the last entry of the node. Let's call the *linearization* of the tree L.

Now let's consider each operation:

- 0 X Y V: Adding a new node X to the tree is equivalent to inserting the pair (X_f, X_l) in L right after Y_f, or right before Y_l.
- 1 X: Removing X and it's subtree is equivalent to removing the interval $[X_f, X_l]$ from L.
- 2 X Y: Moving X and it's subtree as a subtree of Y is equivalent to moving the interval $[X_f, X_l]$ from L right after Y_f, or right before Y_l.
- 3 X V: As each node in a tree has a unique father, the edge connecting the node to its father is also unique. Thus, we can associate the value on the edge from the node K to its father with the node K itself, whatever K would be. Taking this into consideration, XOR-ing all the edges in X's subtree with V is equivalent to XOR-ing all the values associated to the nodes in the interval (X_f, X_l) - Notice the open interval.
- 4 X: Find the XOR-sum of all the edges in X's subtree is equivalent to finding the XOR-sum of all the values associated to the nodes in the interval (X_f, X_l). Knowing that each value appears twice in this interval, we should only consider the nodes indexed with f or the nodes indexed with l.
- 5 X Y: Knowing that $aXORa = 0$, then the XOR-sum of the edges on the path from X to Y is equivalent to the XOR-sum of the values associated to the nodes in the interval $[X_l, Y_f]$. Note that the position of X_l should be smaller that the position of Y_f in L. If this is not the case we could just swap the two numbers.

Each of the above operations can be executed in $O(log(n))$ (where n is the number of elements in L) if we store L in a $Treap$ or a similar data structure. Note that we will insert elements in the $Treap$ by position. Also, at any time we should be able to find out which is to current position of any node in the $Treap$. To do this we will keep a pointer to each node in the $Treap$. Also, each node in the $Treap$ should also store its father. By doing this we will be able to ascend to the root and find out how many elements are to the left of any node, thus finding out its position.