Florida High School Programming Series
2020 Championship Round

Problem Review by Arup Guha

Note: In past years, for this playoff round, I had written all of the questions, but this year, three of us collaborated: Kyle Dencker (Computer Science Teacher at Timber Creek High School, Sharon Barak (Kyle's former student and current UCF Programming Team Member), and myself. Glenn Martin, another UCF Programming Team coach, set up the contest and handled all issues with the online judging system. Kyle Dencker marketed the contest and made sure there were participants. It was his idea to still host the contest online. In the parentheses by each problem I will note the problem author - Arup

Problem A: Toilet Paper Reselling (Kyle Dencker)

Due to having a greater number of new contest participants, we decided to order the problem set, roughly in difficulty order. Thus, this problem was the easiest in the set. As long as Sleazy sells 100 or fewer rolls, his profit is four times the number of rolls he sells. If he sells more than 100 rolls, then for each roll over 100, he loses a dollar. Thus, an if statement to separate these two cases, and implementing the two different formulas will suffice. Most programming languages have a minimum function - one can avoid the if statement by realizing that the amount of money Sleazy earns total is the minimum of 500 and five times the number of rolls he sells. Then, just subtract the number of rolls he sells from this quantity (since each costs a dollar).

Problem B: Generating Usernames (Kyle Dencker)

This is a string manipulation problem. One must properly read in the total number of strings, and concatenate each of the desired pieces from each string and output those pieces. Most languages have a substring function to make this process relatively easy.

Problem C: Group Assignment (Kyle Dencker)

We can simply treat the input as two strings (not integers), and after reading in the strings, we can loop through both, checking to see if there is a '1' in either string at each position. For each position there is at least one '1', we add one to a counter. We take this total value and divide by the length of one of the input strings (both are the same length) and multiply by 100. Unfortunately, this creates a decimal. To cleanly implement the floor function, FIRST multiply the number of slots with at least one '1' by 100, THEN do an integer division by the total number of problems.

Problem D: Board Game (Arup Guha)

There was some discussion amongst the judges if this problem should come fourth or fifth. While the next problem (Class Grades) is simple to code, it requires a somewhat clever observation. This question doesn't require any clever observation, but does require extreme attention to details when coding. The rules of the piece movement are given in the problem and both the size of the board and total number of moves is small enough that all moves can be simulated, one move at a time. One issue is deciding what information needs to be stored, and how to store it. One natural way to store the information is to store the board as an array, and in each slot, store WHICH player is there, or a sentinel value (say -1) for an empty square. This works nicely AFTER each player has moved once, since this doesn't allow us to store all the players at square 0 in the beginning. Though it's overkill, we can overcome this problem by storing an array of lists (or array of arrays), so that at each slot, we just allow more than one player. Another solution is to separately store an array which stores each player's current location. This array has the added benefit of not having to "look" for a player when it is her move. (If we only store the board, whenever someone's move comes up, we would have to loop through the whole board to find that player. But, if we store this information directly, no search is necessary.) If we have this second array, then the board array can just store integers. Simulating a move with or without this array is relatively easy. Do a while loop instead of a for, and only count squares that don't have occupants. In this second system, we have to monitor when board square 0 becomes empty. For the first n-1 players, they MUST move to a square different than square 0, since player n is occupying that square. As soon as player n starts moving, mark square 0 as unoccupied (before she finishes her turn). This way, player n has an opportunity of landing back on square 0 on her first turn, while the rest of the players can not land on square 0 after their first turn.

Problem E: Class Grades (Kyle Dencker)

First we must find the index of the minimum grade. If there are multiple minimum grades, we must find the lowest index storing a minimum grade. Similarly, we must find the highest index storing a maximum grade. We can do each task with a single loop. We can go through our list in order, updating the minimum index ONLY when we see a strictly smaller grade. When we go through oru list again, we update our maximum index when we see a grade that is greater than or equal to our running maximum. Once we have these two indexes, we can calculate the number of swaps necessary to move the minimum index to index 0 and the maximum index to index n-1, where n is the length of the list. One tricky case is if the minimum index is greater than the maximum index. In this case, there is a single swap involving both values that in the regular method gets counted twice. So, this swap must get subtracted out. Here is an illustration of this case:

10 90 5 40

Here, the minimum value is stored in index 2 and the maximum in index 1. Our default answer would be 2 + 2, since the 5 requires two swaps to be put in the beginning and the 90 requires 2 swaps to be placed at the end. But notice that after we do one swap,

10 5 90 40

that both numbers have moved one step closer to their goal, but we only did one swap, not two. So add an if statement to detect this case and if it happens, subtract 1.

Problem F: School Raffle (Arup Guha)

A greedy approach works here. Due to the linearity of expectation, for each individual value, we can calculate our expected value if we were to put in a raffle ticket for that item, and then our total expected value is simply the sum of these separate expected values. Thus, we simply aim to put tickets in for items that give us the highest expected value. If an item has a value of v, and there are a total of k tickets already submitted for that item, if we add our ticket to this item, then our probability of winning the item is $1/(k+1)$, which makes our expected value $v/(k+1)$. Thus, we must calculate this expected value for each item, and then sort the items from most to least value. Finally, add up the first r of these expected values, where r is the number of raffle tickets you submit.

Problem G: Alternate X-Ray (Sharon Barak)

The key to this problem is noting that all points that are on a forward sloping diagonal have a fixed difference between their x and y coordinates while all points that are on a backwards sloping diagonal have a fixed sum of their x and y coordinates. So, for each point, we want to store which "forward sloping line" it's on (it'll always be in between -2,000,000 and 2,000,000) and which "backward sloping line" it's on (also in between -2,000,000 and 2,000,000). We can use two maps/dictionaries, one for forward sloping lines and another for backward sloping lines, mapping each unique index to how many points map to it. Consider the second sample case with the following points: (2, 7), (4, 9), (1, 8), (1, 6), (5, 4) and (2, 8). The sum of the coordinates for each point are 9, 13, 9, 7, 9 and 10. Thus, our map for sums (backwards sloping lines) would store $7 \rightarrow 1$, $9 \rightarrow 3$, $10 \rightarrow 1$, $13 \rightarrow 1$. The differences of these coordinates for each point are -5, -5, -7, -5, 1, and 6. The corresponding map would store $-7 \rightarrow 1$, $-5 \rightarrow 3$, $1 \rightarrow 1$, $6 \rightarrow 1$. It's clear we can take any combination of one line from the first set and one line from the second set. At first it may be tempting just to add the maximum value in the first map and add it to the maximum value in the second map. But this second sample case shows the flaw in that thinking. Notice that both maps have a size of 3, but that there is no X which kids all 6 points. This is because the X formed by all the points with a sum of coordinates of 9 and a difference of coordinates of -5 is centered at (2, 7), which is one of the points. Thus, when we add 3 + 3, we are adding the point (2, 7) twice. Thus, for each pair of line intersections, we must determine if one or our points is at the center of the cross so to speak, and if so, subtract one. To determine this, simply store all the points in a set. When given both the sum and the difference (known from our maps) we can calculate the point itself. Specifically, if $x + y = s$ and $x - y = d$, adding the equation yields that $2x = s + d$ and subtracting yields that $2y = s - d$. Thus, the intersection point is $((s+d)/2, (s-d)/2)$. When determining how many points will be hit for combining the line with sum s and difference d, just check if this point is in the input set. The bounds are probably small enough that one could run a loop through the data to look for the point, but both judge solutions use a set for this look up.

The shorter judge solution just tries all possible pairs of sums and differences. The longer judge solution finds the set of all maximal mappings for sums and the same for differences, and only looks through these sets, noting that whatever the maximal answer is, it must come from one of these sets. The reason for this is that the worst case is that 1 is subtracted from our answer. Thus, the only case that can beat the pair of a max sum and max difference is another max sum and max difference. Subtracting 1 won't make a combination lose to anything else. But, for this problem, this technique doesn't save any run time compared to the shorter judge solution, because one could construct a case where all sums and differences have the same number of values.

Problem H: Jump Conveyor (Sharon Barak)

We can simulate the process of jumping by going to the trampolines as instructed, and if we jump off the main grid, stop. If we return to a trampoline we were at before, we stop, noting that we have spun infinitely. We could repeat this for each trampoline. The problem is that we may take O(n) time, where n is the total number of trampolines, to determine if one trampoline gets in an infinite cycle or not. Thus, in the worst case, if we repeat this procedure for each trampoline, our run time would be O(n^2), which is too slow given that n could be as large as 10^6. The first key is to notice that if we make a big cycle, we should mark ALL of those nodes as being infinite, so that when we are about to check if a trampoline takes us to a cycle, we can see that we've done this one before. This handles the run time issue. But, now we have to think of some interesting cases. It's possible that we "jump into" a cycle. It's also possible that we "jump into" a path that jumps off the list of trampolines. In each case, to save time, we must not redo the previous jump steps. Instead, we must know what our fate will be. Thus, our solution is to do a search from each trampoline. If a trampoline is unvisited (we haven't jumped on it before), start jumping from it, marking each trampoline you reach as visited. Instead of doing this with a boolean value, do this with an integer, where the integer represents which jumping path you took. If we jump off the trampoline list, mark in your list that all the locations with this integer do not lead to infinite jumping. If we jump to a spot that has been previously hit before, do NOT continue jumping. Look to see which jumping path it is, and then also look up the fate of this jumping path. Whatever the fate of this new jumping path you hit, that will be the fate of all trampolines in the current jumping path. Here is a simple example of this situation:

1 -1 1 -2

We can see that the first two trampolines form a cycle. In our visited array, we would note this as follows:

0 0 -1 -1

This means that the first two slots are part of jumping path 0 and we haven't been to the last two slots yet. We would also note in our boolean list that jumping path 0 leads to infinite jumping.

Now, when we search from the third trampoline, it will go to the right marking both squares:

0 0 1 1

Now, when we jump from the fourth trampoline, it leads us to the second one, which has been visited. This second one is part of jumping path 0, and we know that jumping path 0 leads to

infinite jumping, so we mark that jump path 1 leads to infinite jumping as well. Had jumping path 0 not led to infinite jumping, this would have also been the fate of jumping path 1.

Once we complete our search, we will have a list of which jumping path each trampoline belongs to and another list of which jumping paths are infinite. Use these two lists to count how many trampolines lead to infinite jumping!

Problem I: Roads? Where We Are Going, We Don't Need Roads. (Kyle Dencker)

A problem about maintaining connectivity at some minimum cost is usually a minimum spanning tree problem. In this problem, we aim to maximize this cost instead, and instead of maximizing the cost absolutely, we have some extra constraints based on the parity of the road cost. With all of these changes, one might conclude that the minimum spanning tree algorithm would no longer apply. In fact, for other problems such as shortest distance between two vertices in a graph, algorithms such as Dijkstra's can't easily be modified to solve the longest distance problem, for example. This is because the proof of correctness for Dijkstra's breaks down when attempting to apply it to the maximization problem. It turns out that the proof of correctness for all of the minimum spanning tree algorithms does NOT break down when attempting a maximization. The easiest of the proofs to adapt is the proof of correctness of Kruskal's algorithm (however all MST algorithm proofs can be modified to show their applicability to the problem). The proof of Kruskal's stems from this fact: let the set of vertices in the graph be V. Let the sets U and U' be any partition of the set V. It is guaranteed that the minimum edge that connects any vertex in U with any vertex in U' must be in some minimum spanning tree. The proof can be found in any algorithms textbook. Here is one weblink that covers it: http://www.m98.nthu.edu.tw/~s9822506/Kruskal.pdf.

So, let's first consider maximizing edges with even values. Using this proof, if we always prioritize even valued edges over odd edges, then we'll take as many even valued edges as possible before ever even considering an odd edge. If we ever consider an odd edge in the algorithm, this means that none of the remaining even edges add any connectivity between different connected components. Thus, this proves that if we want to maximize the number of even edges, we can simply sort these as being more important than all odd edges and we'll always maximize the number of even edges. Then, within that constraint, if we want to maximize the spanning tree value, the same exchange argument that proves the fact about any partition of vertices can be used is valid. Thus, if two edges have the same parity, we want to consider edges with greater value first. Thus, we can correctly solve the problem by applying Kruskal's algorithm but by sorting the edges in the following order: all even edges before all odd edges, and between even edges, sorting from most to least and between odd edges, also sorting from most to least.

Problem J: Christmas Ornaments (Arup Guha)

This problem was initially designed to be a counting problem to be solved by the inclusion-exclusion principle. Without any constraints on the number of each ornament, the problem reduces to a well-known result covered in introductory combinatorics courses known as combinations with repetition. If one is selecting n items out of k distinct items where there is no limit on each of the distinct items, then the total number of ways to do the selection is just ${}_{n+k-1}C_{k-1}$, where C represents the combination function (${}_nC_k$ represents the item on row n, column k in Pascal's Triangle using 0-based indexing.) But this problem presents a few extra challenges. The first challenge is relatively easy to overcome. If we have a minimum requirement for each ornament, then we have no "choice" so to speak, we can just go buy that minimum number. Doing this subtracts the sum of these minimums from the given target value of ornaments to buy. If doing this subtraction gives us a negative number, there is 0 ways to buy the ornaments. If not, then we have a new equivalent problem with fewer ornaments to buy but maximum constraints on each ornament. Let's consider solving the problem for a maximum constraint on a single ornament. To make this easier to understand, consider the following example with numbers. Let's say we need to buy 20 ornaments from 5 different types, but for one of the types, we can buy at most 7 and the rest are plentiful (we have at least 20 in stock). Without the constraint, our answer would be ${}_{20+5-1}C_{5-1} = {}_{24}C_4$ ways. Now, we must ask ourselves, how many of these combinations that we initially counted have too many of the restricted ornament? We can simply answer that by getting 8 of that ornament immediately, so instead of buying 20, we now want to buy 12 and are free to buy any of the five. Every single one of these combinations has 8 or more of the restricted ornament, meaning that they shouldn't have been counted in the first place. But we know the answer to this question as well: we can do this in ${}_{12+5-1}C_{5-1} = {}_{16}C_4$ ways. Thus, with a single restriction, it follows that the total number of ways to buy the ornaments for this particular case is ${}_{24}C_4 - {}_{16}C_4$ ways. With multiple restrictions, what happens is that we might subtract out a forbidden combination twice. In situations like these, the Inclusion-Exclusion principle indicates how to add back each combination that was subtracted out too many times. The principle requires going through each subset of sets subtracted out. If there are n different sets, the principle requires evaluating the problem for each of the 2^n subsets. In this problem, n was 20, so this evaluation can run in time. With a precomputation of Pascal's Triangle, evaluating each subset becomes very quick. Finally, it's necessary with subtraction to make sure there are no modulo errors, so in some sub-expressions the mod value has to be added to ensure that a non-negative number is being modded.

The easier solution, which was likely implemented by most contestants and is the alternate judge solution requires dynamic programming where the relevant state is how many council person's ornaments have already been given AND how many more ornaments are there to give. For example, f(3, 20) would represent the number of ways we can hand out the ornaments for all the council members except the first three, knowing that we must buy exactly 20 ornaments the rest

of the way. To calculate f(k, t), where k is the index of the current council member and t is how many more ornaments to give, we simply try giving council person k every possible valid # of ornaments within her range. So, let's say her range was 3 to 5, then f(k, t) = f(k+1, t-3) + f(k+1, t-4) + f(k+1, t-5). In each case, we always move onto the next council person, but our new number of ornaments left to give will vary based on how many of the council person k's ornaments we bought. The run-time of this algorithm is the number of council people times the maximum number of ornaments squared because the number of different recursive call is the number of council members times the maximum number of ornaments. For each recursive call, the run time could be as much as the maximum number of different ornaments since the council person could have a wide range to select from. In practice the run time won't get this high for most cases as there are restrictions on most of the council members. Even this maximum is simply 20 x 1000 x 1000 = 20 million, which runs plenty fast.

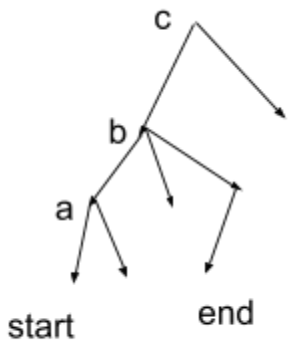
Problem K: Scavenger Hunt (Arup Guha)

The key to solving this problem is to take the description and think about reformulating the solution recursively. If we are at a given node with k outgoing edges, we have a probability of 1/k of going to each of the nodes. What we must determine is, given the expected scores for those future nodes (which our recursive calls will determine), how do we determine the expected score for the given node? Let's first consider the expected path with probability - assuming we will take one particular path. Whatever the expectation from that node, we first want to add 1 for our given node. Then, if our given node has a multiplier, we want to multiply this total by the multiplier, and in this way, it naturally gets applied to all future nodes, similar to how we can evaluate a polynomial in a single for loop or build the binary value of a bit string in a single for loop. Once we have the expected score for one of our k paths, all we have to do is multiply this by 1/k, the probability of taking that path. Next, we simply repeat this for all k paths adding each of these contributions. We can simply take the sum of each of these also, and just divide by k at the end. There are some precision issues, but to make sure these weren't accidentally encountered, during judging, any answer that had a relative error of 1e-6 to the judge solution was accepted.

Problem L: Essential Road Work (Sharon Barak)

First, a straight-forward solution that runs in O(nm) time will be covered, and then a general technique to speed it up will be discussed. Imagine rooting the tree arbitrarily, and running a tree traversal (essentially a depth first search), and for each node, keeping track of its distance (depth) from the root, as well as its parent node (the unique node on the path from the root to it, right before arriving at it). We can keep track of each edge's state, initially setting each edge to be open. When a worker travels, he will go from his start node up the tree some, and then eventually

he'll start descending back down the tree to his destination, changing the open/close state of each of the edges he travels. The problem is knowing when he starts descending back down. If he starts too early, then he won't arrive at his destination, if he starts descending too late, he'll retrace steps he just took (which isn't allowed). Here is a small picture of what is being described:



In this picture, where the root of the tree is c, imagine a worker traveling from start to end. If they started heading back down the tree at a, then won't ever get to end. If they start heading to end after reaching c, they'll travel the edge from b to c going up and then again going back down, which is redundant. The worker must start going back down the tree when arriving at b. For trees, the node b is defined as the "lowest common ancestor" (LCA) of start and end. It's furthest node from the root of the tree such that both start and end are both in its same subtree. In a tree, between any two nodes, the shortest path is always through the LCA of the two nodes. It is possible that one node is a direct ancestor of another, in which case you never have to start turning back down the tree.

One (slow) way to compute the LCA of two nodes in a tree, if you already know the depths of both nodes is to take the node that is further down in the tree, and start crawling up the ancestral path, one by one, until you arrive at the same level of the other node. Now, in lock-step, start both nodes marching up the tree, one step at a time until you arrive at the same node. In this picture, start and end are already both at depth 3. After crawling up lock-step twice, we'll arrive at node b as the LCA. In our slow code, we can simply just switch the state of each edge as we crawl up.

Unfortunately, crawling up the tree one step at a time may take up to O(n) work, where n is the number of nodes in the tree to trace through a single worker's path. With m workers, repeating this work m times gives us a run time of O(nm), which will not pass under the given time limit.

Thus, we need to do two things to speed this up:

(1) Crawl up the tree faster, skipping steps.
(2) Find a way to calculate what we want for every node without toggling each individual edge as we travel. Instead, just given the LCA of two nodes, we must be able to do a quick calculation that will allow us to calculate the end answer.

A technique called binary lifting will allow us to find the LCA of two nodes of a tree in O(log n) time, where there are n nodes in a tree. The full details are too much to include here, but here is a tutorial on how it works: https://cp-algorithms.com/graph/lca_binary_lifting.html. The gist of the idea is that at each node, instead of just storing its parent, we store the node that is every power of 2 above. So, if a node is at level 22, in the node itself we store links to its ancestors (nodes on the path from it to the root) that are at levels 21, 20, 19, 17, 13 and 5. The extra storage needed is O(log n) per node, so instead of storing the tree in O(n) space, we need O(log n) space and we need to pre-compute all of these ancestors. (This isn't too hard to do because once we know who is at level 13, for example, then level 13's lifting chart already tells me which node is 8 above it.) Now, once we have all this information, we use the bitwise representation of how many levels to go up the tree to jump up so that it'll take us at most log n jumps to get to where we want to get. For the LCA algorithm, we will still take the deeper node and jump up to the same level as the other node. After that, we'll binary search how far to jump up, stopping at the node right before the LCA (the most we can jump without hitting a common ancestor), and continue to do this until we jump 1 away to prove we are at the LCA.

Now for the second part. Once we know where the LCA of two nodes is, we want to "mark" two paths: one from the LCA to the start node and one from the LCA to the end node. This becomes only one path if the LCA is one of the two nodes. For each path we mark, we add 1 to the lower endpoint and add 1 from the LCA. Then, once we have these values, we can run a single depth first search on the tree. In doing the DFS, we will keep a score of each node, which will equal the score for the edge from the node's parent to it. The score of each node is simply the sum of the original scores of all the nodes in its subtree. Thus, while running the DFS, when descending down an edge, the score for that edge will be the sum of the values of all the nodes in the subtree BELOW that edge. If this sum is even, the edge(road) is open, if it's odd the edge(road) is closed.