Code Jam 2020 - Round 1A

Analysis: Pattern Matching

Test Set 1

In Test Set 1, each pattern forces our answer to have a certain suffix, and we first need to check whether the patterns introduce conflicting requirements for that suffix.

Consider the letter strings coming after the initial asterisk in each pattern. We can find the longest of those strings (or any longest, if there is a tie); call that string L. Then at least one answer exists if (and only if) every other string is a suffix of L; note that we are considering L itself to be a suffix of L. We can check each other string against L by starting at the ends of both strings and stepping backwards through them in tandem until we find a discrepancy or run out of letters to check. If we ever find a discrepancy, then the case has no answer, but otherwise, we know that L itself is an acceptable answer.

Test Set 2

In Test Set 2, we can divide the patterns into (1) patterns that start with an asterisk, (2) patterns that end with an asterisk, and (3) patterns with an asterisk in the middle.

A type (1) pattern requires the output word to have a certain suffix, just as in Test Set 1. A type (2) pattern requires the output word to have a certain prefix. A type (3) pattern introduces both of these requirements, and we can split it into a suffix requirement and a prefix requirement, and then handle those separately.

Then, we can apply our strategy from Test Set 1 twice: once for the prefix constraints (with the algorithm modified to compare prefixes instead), and once for the suffix constraints. We can concatenate the two results together to obtain a valid answer that is certainly short enough (since it can be at most 99+99 characters long).

Test Set 3

We can generalize the idea above into a solution for Test Set 3. Each pattern p in Test Set 3 also prescribes a prefix of the output word (the prefix of p up to the first asterisk) and a suffix of the output word (the suffix of p starting after the last asterisk). If we allow empty prefixes and suffixes, we get exactly one of each for every pattern. We can handle those in the same way we did for Test Set 2, ending up with a prefix P and a suffix S for the output as long as we do not find a discrepancy in either phase.

However, for patterns that have more than one asterisk, we can also have a middle part, which imposes a new type of requirement. Suppose we parse the parts between the asterisks of a pattern so that X is the prefix up to the first asterisk, Y is the suffix after the last asterisk, and M_1, M_2, ..., M_k are the strings in between the asterisks, in order. After checking that X is a prefix of P and Y is a suffix of S, as before, all that remains to ensure is that the pattern $M_1*M_2*...*M_k$ is present somewhere within the output word, strictly between P and S.

Let us call $M_1M_2...M_k$ — that is, the word that occurs in between the first and last asterisks with any other asterisk removed — the *middle word*. If we make sure a pattern's middle word occurs in the output word outside of P and S, then we fulfill the extra requirement. We can then build a full valid output then by starting with P, then adding the middle word of every pattern in any order, then appending S. We make sure to correctly handle words with a single asterisk or only

consecutive asterisks by making their middle words the empty string. Since each middle word contains at most 98 characters, and the prefix and suffix contain at most 99 characters each, the output built this way has at most $99 \times 2 + 50 \times 98$ characters, which is within the limit of 10^4.

Code Jam 2020 - Round 1A

Analysis: Pascal Walk

Test set 1

The answer for **N** ≤ 500 can always be constructed by walking along the leftmost positions of the rows of the triangle: (1, 1), (2, 1), ..., (**N**-1, 1), (**N**, 1).

We can handle the case **N** = 501 by making a detour to pick up the 2 in the third row before returning to the left side. That is, our walk takes us through the positions (1, 1), (2, 2), (3, 3), (3, 2), (3, 1), (4, 1), ..., (498, 1).

Test set 2

With only 1000 possible test cases to consider, we can find an answer to each one before submitting. One way to do this is to walk over the triangle in e.g. a breadth-first or random way, taking care never to reuse a position or visit more than 500 positions. We can check our cumulative sum at each position, and whenever we encounter a sum that we have never seen before, we record the sequence of positions that we used to obtain it. When our sum exceeds 1000, we backtrack in our search or start over with a new random path, and so on, until we have an answer for every possible **N**. It's difficult to prove a priori that this will work, but we can be optimistic given that the top few rows of the triangle have many small values to work with. Indeed, this method finds a full set of solutions very quickly.

An alternate method is to observe that the positions immediately to the right of the leftmost positions — named (x, 2) for each x ≥ 2 — are 1, 2, 3, 4, 5, and so on. We can move from the top position in the triangle down to the 1 at position (2, 1), then follow that line to the 2 at (3, 2), the 3 at (4, 3), etc., until our next move would cause the cumulative sum to exceed our target. Then, we can instead make a move to the left to reach a 1 on the left edge of the triangle, and then proceed downward along that edge, taking as many extra 1s as we need. Since the sum of the first 45 natural numbers is over 1000, we can be sure that we will never need to take more than 45 of these extra 1s, or visit more than 45 of the positions along this line of natural numbers. This ensures we never use more than 90 positions in total.

Test set 3

There are various ways to solve the third test set, but one of our favorites takes advantage of a special property of Pascal's triangle: the entries in the r-th row (counting starting from 1) sum to 2^{r-1}. This is to be expected from the way in which each row is constructed from the previous one: each element in a row contributes to two elements in the next row, so the sum doubles with each row.

This observation suggests a strategy for constructing any **N** that we want: write **N** in binary, and then look through that representation, starting from the least significant bit. If the r-th least significant bit (counting starting from 1) is a 1, our path should consume all of the elements in the r-th row. If the r-th least significant bit is a 0, we should skip that row somehow. We only need the first 30 rows of the triangle, since the 31st row's sum is 2^{30}, which is larger than the maximum possible **N** of 10^9. Even if we used every element in these first 30 rows (which we would never need to), that would only use 465 positions, which is within the limit of 500.

This doesn't quite work as written, though, because our path must be continuous, and so there is no way to skip a row. But we can get close via the following variant: proceed down one side of

the triangle (that is, the diagonal line of 1s) as long as we keep encountering 0 bits. Whenever we encounter a 1 bit, we send our path horizontally across the corresponding row, sending us to the other side of the triangle. This almost gets us the number we want, but we probably overshoot because we have consumed some extra 1s from the rows corresponding to 0 bits. We can be sure that there are fewer than 30 of those, though, since we visit at most 30 rows.

So, we can use that variant, but instead of constructing **N**, we construct **N**-30 instead. Once we finish, we can tack on additional 1s from our current edge of the triangle until we reach **N**. (We can handle cases with **N** ≤ 30 specially, and just go down one edge of the triangle.)

There are other less elaborate ways to solve this test set, though! One is to walk down the centers of the rows (e.g., going from (1, 1) to (2, 1) to (3, 2) to (4, 2) to (5, 3)...). This is where the largest numbers live, so we will grow our sum as rapidly as possible. At some point, though, when moving downward would make our sum grow too much, we can instead move to a one step away from the center and continue zigzagging downward, and so on. Eventually we will reach the edge full of 1s, and we can take as many more as we need. We must be careful not to increase the sum so fast that we will not be able to "escape" to the edge of 1s. Nor should we reach the edge too early, since we can only take so many 1s. However, it's not too difficult to get this method to work.

Code Jam 2020 - Round 1A

Analysis: Square Dance

Test Set 1

Test Set 1 can be solved by simulating the competition:

1. Iterate through all the cells of the floor. For each cell, if there is a dancer in that cell:
    - Add their level to the interest level of the competition.
    - Loop through their row and column to find all of their compass neighbors.
    - Count the number of their compass neighbors, and sum up their levels.
    - If the average level of the neighbors (calculated as sum of their levels divided by how many neighbors there are) is greater than the level of the current dancer, add this dancer to an elimination list.
2. Go through the elimination list and eliminate the dancers.
3. If the elimination list is empty, we are done. Otherwise, start again from step 1.

Each iteration of the above algorithm has a time complexity of O(**R**×**C**×(**R** + **C**)), and there are as many iterations as there are rounds in the competition. The number of rounds in a competition cannot be larger than the total number of dancers, **R** × **C**. So the total time complexity is O($\mathbf{R}^2$×$\mathbf{C}^2$×(**R** + **C**)).

Test Set 2

Let's optimize our Test Set 1 solution — specifically, the following parts of it:

- For each round, we iterate through all **R**×**C** cells and check whether each dancer is eliminated.
- For each dancer, we may have to iterate through a significant part of their row and column in order to find their compass neighbors.

To improve the situation described in the first point above, we can observe that for i > 1, if dancer d is not eliminated in round i-1 and they are eliminated in round i, their set of compass neighbors must have changed in between. For that to have happened, at least one compass neighbor of d must have been eliminated in round i-1.

Using this observation, instead of checking every dancer in round i, we can limit the check to the compass neighbors of those eliminated in round i-1. Since a dancer has at most 4 compass neighbors at the time of elimination, the length of the list of candidates to check in round i is at most 4 times the length of the elimination list for round i-1. The sum of the lengths of all elimination lists is at most **R** × **C**; therefore, the sum of the lengths of all lists of candidates to check is at most 4 × **R** × **C**. With the initial check of all dancers, this means that we do O(**R** × **C**) checks overall with this improvement.

To optimize finding compass neighbors, notice that when we remove a dancer, their eastern neighbor becomes the eastern neighbor of their western neighbor and vice versa. The same thing happens to the southern and northern neighbors. So, instead of finding the neighbors each time we need them, we can maintain the specific location of the neighbor in each direction for each dancer, and update their locations in constant time whenever we eliminate one. This is effectively like keeping a linked list to represent each row and column, and this structure allows us to remove and find neighbors in constant time.

Since we do O(**R** × **C**) checks thanks to the first optimization, and each check can be done in constant time thanks to the second optimization, this yields an O(**R** × **C**) time algorithm that solves the problem fast enough for Test Set 2.