Code Jam 2020 - Round 1C

Analysis: Overexcited Fan

Test Set 1

In Test Set 1, we can try every possible path we can take. During any given minute we have 5 options: walk a block in one of the 4 directions, or stay still. Since Peppurr's tour is at most 8 minutes long, we can only walk for at most 8 minutes. That is at most 5^8 possibilities, which is a pretty small number for a computer. For each combination, we simulate our own path and Peppurr's path, recording any encounters. After trying all possibilities, the solution is `IMPOSSIBLE` if we did not record any encounters, or the earliest time of those if we did.

Test Set 2

In Test Set 2, just as in Test Set 1, Peppurr's tour will remain within a single north-south street. Notice that if we want to meet Peppurr at an intersection (a, b), the order in which we walk the blocks doesn't matter as long as we walk east a times more than west and north b times more than south. So we may assume that we can finish all of our eastward walking, for example, before walking in any other direction. This means an optimal strategy can begin with **X** blocks of walking east. After that, we are in the same north-south street as Peppurr, so we can walk towards the tour until we meet or we are just 1 block away, in which case we need to stand still for 1 minute to avoid crossing paths with the tour in the middle of a block.

Test Set 3

For Test Set 3, we can simulate Peppurr's tour. If after R minutes it is X_R blocks to the east of us and Y_R blocks to the north (X_R and Y_R can be negative to represent being west or south), we only need to check whether we can reach that intersection in R minutes. Fortunately, this is easy to check: the intersection is reachable in R minutes or less if and only if $|X_R| + |Y_R| \leq R$. That is, the intersection must be within an L1 distance (also known as Manhattan Distance) of R.

Therefore, we can solve the problem by simulating Peppurr's path, and for the i-th intersection visited, check if it's reachable in i minutes. If it is, then i is the answer; otherwise, we need to keep looking. If none of the intersections is reachable within the required time, we answer `IMPOSSIBLE`.

Code Jam 2020 - Round 1C

Analysis: Overrandomized

Test Set 1

In Test Set 1, the range for possible M values is so small compared to the number of records that each combination (M_i, N_i) has a somewhat large probability 1 / (99 × M_i) of appearing as a particular record, and an even larger probability of being present as at least one record.

Suppose there exists at least one record with $M_i = N_i = x$ for each x in the range 1 through 9. From the record with $M_i = N_i = 1$ we know that the only letter in $\mathbf{R_i}$ represents 1. Then, from all the records with $M_i = 2$, we can discard the ones where $\mathbf{R_i}$ represents 1. The leftovers must be the ones with $M_i = N_i = 2$, and in those, the only letter in $\mathbf{R_i}$ represents 2. In general, after we have decoded the letters for 1 through x, we can take the records with $M_i = x + 1$ and discard the ones with letters in $\mathbf{R_i}$ that are already assigned, and the $\mathbf{R_i}$ values of the remaining records will contain the letter that should be assigned to x + 1. Finally, the only remaining unassigned letter should be assigned to 0.

This process works as long as records with $M_i = N_i = x$ exist for each x in the range 1 through 9. The probability of that happening is hard to calculate, but the least likely of those combinations is $M_i = N_i = 9$, with a probability of only 1 / (99 × 9) per record. The probability of that appearing at least once in 10000 records is greater than 99.999%. The smaller values of x have even higher probability. Of course, the probability of all 9 coexisting is smaller than that, and the probability of that happening in all 10 cases is even smaller, but still decent enough. In addition, there is a really small probability that the letter representing 0 doesn't appear at all in the input, but if that were the case, no algorithm could find it. Given that this is a Visible Verdict test set, we can give the solution a try, and confirm that it passes.

There are additional heuristics we could add to the algorithm. For example, if we can't find the value for 9 in this way and we need to distinguish the two remaining letters to assign to 9 and 0, just having a record whose $\mathbf{R_i}$ starts with one of the letters is enough to know that that one should be a 9, since 0 cannot be a leading digit. This further increases the probability of the method working. We can add more and more heuristics to cover the remaining cases, but at some point it's easier to just try something more general.

Test Set 2

In Test Set 2, the probability of M_i being a single digit is small, and we cannot rely on that happening, let alone several times and with extra conditions. However, we can treat records that have M_i and $\mathbf{R_i}$ of the same length similarly, by simply using their first letters and then using those (digit, letter) pairs as we did in the solution for Test Set 1.

Test Set 3

At this point, it seems like we may have to throw all of the above insights away, because they are all predicated on knowing M_i. However, the "use the leading digit/letter" insight we used to solve Test Set 2 is actually the first step toward solving Test Set 3 as well.

In Test Set 3, each record's information comes from a single integer, not two, so we cannot use the association between the two parts as before. We can start by checking the distribution used

to generate the only piece of information we have. The probability of any particular N_i being equal to x is the sum of 10^{-16} / y for y in [x, 10^{16} - 1], which can be approximated by 10^{-16} (ln 10^{16} - ln x). In other words, the probability is decreasing in x. Because of this, leading digits are more likely to be smaller than larger. Moreover, even though the decrease in probability between the actual result being x and x + 1 is small, the decrease in probability between the leading digit being d and d + 1 is large, because it aggregates the differences between the probability of dS being more than the probability of (d+1)S for each possible suffix S. This is a version of Benford's law.

Therefore, a possible solution is to calculate the frequency with which each letter appears as a leading digit, and assign the highest frequency to 1, the second highest to 2, etc. The only letter that never appears as a leading digit, and therefore has the minimum frequency, should be assigned to the digit 0.

Code Jam 2020 - Round 1C

Analysis: Oversized Pancake Choppers

Test set 1

In the first test set, we are only asked to produce 2 or 3 equal slices. Let us consider these cases separately.

For **D**=2, if we already have two equal slices, then we don't need any cuts. If no two slices are equal, then we can cut any slice into two equal halves with one cut.

Similarly, for **D**=3, if we already have three equal slices, then we don't need any cuts. We can also cut any slice into three equal slices with two cuts. The extra case to consider is whether we can do it with a single cut.

If we do only 1 cut we end up with **N**+1 slices: **N**-1 original slices and two new slices. Three of them need to be of the same size, so this size has to be equal to the size of at least one uncut slice. We can try all possibilities (up to **N**) for the target size and all possibilities (up to **N**) for which slice we cut. If the slice p to be cut is not larger than the target size s, we disregard the case. Otherwise, we cut p into a part of size s and another part of size $\mathbf{A_p}$-s. Then, if there are 3 slices of size s in the set of **N**+1 slices, it is possible to do it with one cut. If that doesn't happen for any of the considered possibilities, then we definitely need two.

Test Set 2

Let's define a *fully-usable slice* as a slice which we can use fully to produce the slices we need, either by cutting it into 2 to **D** equal sized slices, or just using it in its entirety as it is. That is, a fully-usable slice is a slice that will leave no further leftovers.

Here is a key observation: for every slice we will produce, we will need to use one cut, except possibly for one slice cut from each fully-usable slice we use. That is, we will need **D**-K cuts to produce **D** equal slices, where K is the number of slices used fully.

For example:

- It is always possible to produce **D** equal slices by cutting any original slice with **D**-1 cuts (K=1).
- The best possible case is when we already have **D** equal original slices, because we make 0 cuts (K=**D**).

Also notice that we never have to consider K=0, since K=1 is always possible by cutting one original slice into equal pieces, and we want the maximum possible K.

By the observation above, the final size of our produced slices (hereafter the *target size*) is going to be one of the original sizes (from one of the slices we fully use) divided by an integer between 1 and **D**. Therefore, we have to check at most **N**×**D** possible target sizes. With any other size, we would have 0 fully-usable slices.

For each such target slice size, we should do the following:

- First, we ensure that we can actually use it: if the total number of slices of this size that can be produced by cutting all original slices is less than **D**, then, obviously, this size is not useful for us. A slice of size $\mathbf{A_i}$ can be used to produce up to floor($\mathbf{A_i}$/s) slices of target size s.

- Then, we need to find all the fully-usable slices: their size is evenly divided by the target slice size, with no remainder.
- Now, since we need to maximize the number of original slices that are fully-usable, we can use a greedy approach and take those slices one by one in non-decreasing order of size, until we have as many fully-usable original slices as possible (that is, taking the next one would cause us to produce more than **D** target slices). If we use up all fully-usable original slices, we could use the other non-fully-usable original slices in any order.
- As per our prior observation, each fully-usable original slice gives us one "free" target slice; all other target slices will need a cut each to be produced. That is, the total number of cuts for the current target slice size is **D** minus K, the number of fully-usable original slices we use.

The part above can be done in O(**N**) time if we sort the $\mathbf{A_i}$s once (in non-decreasing order) at the beginning, resulting in an O($\mathbf{D}\times\mathbf{N}^2$) time complexity for the overall algorithm.

Test Set 3

First of all, notice that we can precompute the largest possible target slice size in O(log(max($\mathbf{A_i}$))×**N**) time with a binary search on the target size. Then, we can save some time by not considering any target slice sizes that are greater than the calculated limit.

Now, as in the solution for Test Set 2, we iterate through the $\mathbf{A_i}$s (in non-decreasing order) and all numbers of cuts c (1 to **D**). Instead of doing an additional pass through **N** original slices as in the solution for Test Set 2, we can just mark this original slice as a fully-usable slice for target size $\mathbf{A_i}$/c. To do that efficiently, we use a dictionary (ideally implemented as a hash table) where the keys are valid target sizes, and the values are tuples containing the number of fully-usable slices found so far for that target size, and the number of target slices produced from them. Notice that the keys are fractions; to ensure that we do not represent the same fraction in multiple ways, we can use an algorithm to find greatest common divisors and ensure that all fractions are reduced.

For each target size $\mathbf{A_i}$/c we add 1 and c to the corresponding dictionary values for key $\mathbf{A_i}$/c (assuming the default value for an unset key is zero). If after this operation the number of produced target slices of size $\mathbf{A_i}$/c would exceed **D**, we should just not consider that slice as fully-usable for this case.

Then we can choose the maximum possible number M of fully-used slices across all valid target slices, and the result is **D**-M. This improves the time complexity of the algorithm to O(**D**×**N**).