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**).