Code Jam 2017 - Round 2

Analysis: Fresh Chocolate

Fresh Chocolate: Analysis

The main initial observation to have is that the group sizes are only important modulo **P**. Then, we can just take the smallest possible equivalent size for each group, and further assume that all of them are within the range [1,**P**]. Moreover, you may find it easier to map them all to the range [0,**P**-1], using the modulo operation, as long as you don't mind working with 0, which is not a real group size.

We can see now that a test case with **P**=2 can be described by 2 integers: the number of groups with odd size, and the number of groups with even size. Similarly, each test case can be described by a tuple of exactly **P** integers a_0, a_1, ..., $a_{\mathbf{P}-1}$, where a_i is the number of groups of size equal to i modulo **P**.

The groups counted in a_0, i.e., those with a group size multiple of **P**, are the simplest. No matter when they are brought in for the tour, the number of leftovers after they are given chocolate is the same as before. Therefore, their position won't change how many of the other groups get all fresh chocolate. This implies that it's always optimal to greedily choose a position for them where they get all fresh chocolate, and that we can accomplish that by putting them all at the beginning. Since the starting group gets fresh chocolate, a stream of groups with sizes that are multiples of **P** will all get fresh chocolate and will leave no leftovers for the next group. That is, we can solve the problem disregarding a_0, and then add a_0 to the result. After this simplification, there are only **P**-1 numbers left to consider.

Small dataset

For the Small dataset, there are only two possible values for **P**, so we can consider them separately.

For **P**=2, there is only one number to consider: a_1. Since all groups are equivalent, there are no decisions to be made. All odd-sized groups will alternate between getting all fresh chocolate and getting a leftover piece. Since the last group will get fresh chocolate if there is an odd number of such groups, the number of groups that get fresh chocolate is ceiling(a_1 / 2).

For **P**=3, there are two numbers to consider: a_1 and a_2. Intuitively, we should pair each 1-person group with a 2-person group to go back to having no leftovers as soon as possible, and that's indeed optimal: start with an alternation of min(a_1, a_2) pairs, where we get min(a_1, a_2) added to our result, and then add pairs, of which min(a_1, a_2) will get all fresh chocolate. Then add all the remaining M = |a_1 - a_2| groups (all of size 1 or 2 modulo **P**, depending on which type had more to begin with). Of those, ceiling(M / 3) will get all fresh chocolate. This last calculation is similar to what we saw for odd-sized groups for **P** = 2. We will prove the optimality of this strategy below.

Large dataset

The remaining **P**=4 case is a bit more complicated than just combining our insights from the **P**=2 and **P**=3 cases. To formalize our discussion, and to prove the correctness of our Small algorithm, we will introduce some names. Let us call a group fresh if it is given all fresh chocolate, and not fresh if it gets at least one leftover piece. Also, we will call a group with a number of people equal to k modulo **P** a "k-group".

Given some fixed arrival order, let us partition it into blocks, where each block is an interval of consecutive groups that starts with a fresh group and doesn't contain any other fresh groups. That is, we start a new block right before each fresh group. The problem is now equivalent to finding the order that maximizes the number of blocks. A group is fresh if and only if the sum of all people in the groups that preceded it is a multiple of **P**. This implies that reordering within a block won't make any other blocks invalid, so it won't make a solution worse (although it could improve it by partitioning a block into more than one block). Also notice that, with the possible exception of the last block, reordering blocks also doesn't alter the optimality of a solution.

Suppose we have an optimal ordering of the groups, disregarding 0-groups as we mentioned above. First, we can see that if a block contains a k-group and a different (**P**-k)-group, then it only contains those two: otherwise, we can reorder the block putting the two named groups first, and since the sum of people of the two groups is a multiple of **P**, that partitions the block further, which means the original solution is not optimal. Second, let us show that it is always optimal to pair a k-group with a (**P**-k)-group into a block. Assume an optimal order with a maximal number of blocks consisting of a k-group and a (**P**-k)-group. Then, suppose there is a k-group in a block A with no (**P**-k)-group and a (**P**-k)-group in a block B with no k-group. We can build another solution by making a new block C consisting of the k-group from block A and the (**P**-k)-group from block B, and another block D consisting of the union of the remaining elements of blocks A and B. If D doesn't sum up to a multiple of **P**, that implies that either A or B didn't to begin with, so we can just place D at the end in place of the one that didn't. C can be placed anywhere in the solution. This makes a solution with an additional pair that is also optimal, which contradicts the assumption that a k-group and a (**P**-k)-group existed in separate blocks. This proves that, for **P**=3, it is always optimal to pair 1-groups and 2-groups as we explained above.

For **P**=4, the consequence of the above theorem is that we should pair 2-groups with themselves as much as possible and 1-groups with 3-groups as much as possible. This leaves at most one 2-group left, and possibly some 1-groups or 3-groups left over, but not both. Since we need four 1-groups or four 3-groups to form a non-ending block, and we can use a 2-group to form a non-ending block with only two additional groups of either type, it is always optimal to place the 2-group first, and then whatever 1-groups or 3-groups may be left. Overall, the solution for **P**=4 is a_0 as usual (singleton blocks of 0-groups), plus floor(a_2 / 2) (blocks of two 2-groups), plus min(a_1, a_3) (blocks of a 1-group and a 3-group), plus ceiling((2 × (a_2 mod 2) + |a_1 - a_3|) / 4) (the leftover blocks at the end).

Notice that even though the formality of this analysis may be daunting, it is reasonable to arrive at the solution by intuition, and the code is really short. If you have a candidate solution but you find it hard to prove, you can always compare it against a brute force solution for small values of N to get a little extra assurance. Or there is also ...

A dynamic programming solution

The insight that a case is represented by a **P**-uple is enough to enable a standard dynamic programming solution for this problem. Use **P**-uples and an additional integer with the number of leftovers as current state, and recursively try which type of group to place next (there are only **P** alternatives). Memoizing this recursion is fast enough, as the domain is just the product of all integers in the initial tuple, times **P**. When the group types' sizes are as close to one another as possible, we get the largest possible domain size, which is $(\mathbf{N}/\mathbf{P})^{\mathbf{P}} \times \mathbf{P}$. Considering the additional iteration over **P** possibilities to compute each value, the overall time complexity of this approach is $O((\mathbf{N}/\mathbf{P})^{\mathbf{P}} \times \mathbf{P}^2)$. Even for the largest case, this is almost instant in a fast language, and gives plenty of wiggle room to memoize with dictionaries and use slower languages. A purposefully non-optimized implementation in Python takes at most half a second to solve a case in a modern machine, so it finishes 100 cases with a lot of time to spare. Moreover, just noticing that groups of size multiple of **P** can optimally be placed first makes the effective value of **P** be one less, which greatly improves speed and makes the solution solve every test case instantly.