Kick Start 2020 - Round A

Analysis: Plates

From the constraints, we can see that regardless of the test set, $1 \le \mathbf{K} \le 100$. i.e., $1 \le \mathbf{P} \le 100*\mathbf{N}$.

Test set 1

For this test set, we see that $1 \le \mathbf{N} \le 3$. So, we can check for every possible combination of taken plates across all stacks and output the maximum sum. For example, if **N** = 3 and for any given values of **K** and **P**, generate all possible triples (S_1, S_2, S_3) such that $S_1+S_2+S_3 = \mathbf{P}$ and $0 \le S_i \le \mathbf{K}$. Note: S_i is the number of plates picked from the i-th stack.

This can be done via recursion and the total time complexity is $O(\mathbf{K}^{\mathbf{N}})$ which abides by the time limits.

Test set 2

The solution we had for test set 1 doesn't scale given that N now is at most 100. In order to tackle this test set, we use Dynamic Programming along with some precomputation.

First, let's consider an intermediate state *dp[i][j] which denotes the maximum sum that can be obtained using the first i stacks when we need to pick j plates in total*. Therefore, dp[**N**][**P**] would give us the maximum sum using the first **N** stacks if we need to pick **P** plates in total. In order to compute dp[][] efficiently, we need to be able to efficiently answer the question: *What is the sum of the first x plates from stack i?* We can precompute this once for all **N** stacks. Let *sum[i][x] denote the sum of first x plates from stack i*.

Next, we iterate over the stacks and try to answer the question: *What is the maximum sum if we had to pick j plates in total using the i stacks we've seen so far?* This would give us dp[i][j]. However, we need to also decide, *among those j plates, how many come from the i-th stack?* i.e., Let's say we pick x plates from the i-th stack, then *dp[i][j] = max(dp[i][j], sum[i][x]+dp[i-1][j-x])*. Therefore, in order to pick j plates in total from i stacks, we can pick anywhere between [0, 1, ..., j] plates from the i-th stack and [j, j-1, ..., 0] plates from the previous i-1 stacks respectively. Also, we need to do this for all values of $1 \le j \le \mathbf{P}$.

The flow would look like:

```
for i [1, N]:
   for j [0, P]:
      dp[i][j] := 0
         for x [0, min(j, K)]:
            dp[i][j] = max(dp[i][j], sum[i][x]+dp[i-1][j-x])
```

If we observe closely, this is similar to the [0-1 Knapsack Problem] with some added complexity. To conclude, the overall time complexity would be O(**N*****P*****K**).