Subtask 1 for 10 p.

The intended solution here is any sort of exponential solution that tries all possible cases, checks whether they are valid and then finds the maximum out of all of those.

Initial observations

Here we will examine the structure of the problem, so we know what we're actually trying to solve.

First, let us note that we refer to participants as "nodes", and to participant-target pairs as "edges". Furthermore, we shall also refer to a participant's target as its "child" and to the reverse as its "parent". This is a slight abuse of terminology, as we are not working with a forest but makes it easier to talk about things. Finally, we refer to successful participants as "chosen" or "taken". Note that the type of graph described is a set of independent directed cycles of length at least two.

Second, let us make an observation that is not strictly necessary to solve the problem but it reduces the number of cases that we need to consider, thus simplifying the code and improving the run time by a constant factor. There always exists an optimal solution where each chosen node keeps its badge. Suppose we have an optimal solution where this is not the case; let A be a chosen node which doesn't have its badge. Then A has K random badges to substitute for it. There are three cases for A's badge. First case: nobody has it, then we simply give it back to A. Second case: A's parent has it, then A trades its K random badges for its own badge with its parent. Third case: a random node has A's badge, then A trades one of its random badges for its badge with than node. Therefore, in all three cases A can get its badge back, while preserving the solution, so there is always an optimal solution where each node we take keeps its own badge.

Finally, let us consider the four possible cases for each node A. First case: A is not taken and its badge us free to go to a random node. Second case: A is not taken and its badge goes to its parent (it must be the case that A's parent is taken as otherwise this is pointless). Third case: A is taken using its own badge and K random badges. Fourth case: A is taken using its own badge and its child's badge (it must be the case that A's child is not taken, as shown previously). Now, note that there is a K-to-One relation between nodes in cases one and three (some nodes in the first case may have their badges go to no-one, and not be part of a relation). This relationship is also not related to the order of the nodes. This suggests keeping track of some sort of balance of free random badges. Furthermore, note that there is a One-to-One relation between nodes in cases two and four. Also, they must be adjacent nodes. This means that we can directly process them at once or have a binary sort of "balance".

Subtasks 2 and 3 for 30 p.

Let us first understand the additional constraints. The first part is saying that all nodes are part of single cycle of length N. The second part is saying that there exists a "worthless" node – one we would never need to choose in an optimal solution. Therefore, its outgoing edge is also useless. So, we can threat the graph as a single path. Finally, the $P_0 = 0$ part means that the first node of the path is node 0.

All of this, together with our initial observations, suggests that we can use a simple dynamic programming approach. Obviously, we need to keep track of the position and, as previously noted, the "balance" of free random badges. Note that, since the first-third case relation does not depend on the order of the nodes, it could be the case that we first need to borrow K badges and pay them off later one. Therefore, this dimension can take negative values too. Values from $-NK$ to $+N$, in fact.

One way to handle the second and fourth cases, is to use them separately. We use an extra binary dimension to mean that we have last used case four. Therefore, we set it to true when we use case four at a node, thus borrowing its child's badge. Then, we reset it to false at the child by repaying the badge.

$$dp[pos][balance][0] = \max\begin{Bmatrix} dp[pos-1][balance-1][0], \\ dp[pos-1][balance][1], \\ dp[pos-1][balance+K] + value[P_{pos}] \end{Bmatrix}$$

$$dp[pos][balance][1] = dp[pos-1][balance][0] + value[P_{pos}]$$

$$ans = \max\{dp[N-1][balance][0]\}_{balance \geq 0}$$

Another option is to use the extra dimension to simply mean that we've the node is chosen (but we have already borrowed K random badges). Then, we can get them back at the next node by it giving its badge to its parent, but it is also valid not to do that:

$$dp[pos][balance][0] = \begin{Bmatrix} dp[pos-1][balance-1][0], \\ dp[pos-1][balance-K][1] \end{Bmatrix}$$

$$dp[pos][balance][1] = \begin{Bmatrix} dp[pos-1][balance+K][0] + value[P_{pos}], \\ dp[pos-1][balance+K][1] + value[P_{pos}] \end{Bmatrix}$$

$$ans = \max\{dp[N-1][balance][extra]\}_{balance \geq 0, extra \in \{0,1\}}$$

It is pretty clear that the two formulations above reduce to each other. In fact, we can skip the extra dimension and directly handle the second and fourth cases at once:

$$dp[pos][balance] = \max\begin{Bmatrix} dp[pos-1][balance-1], \\ dp[pos-1][balance+K] + value[P_{pos}], \\ dp[pos-2][balance] + value[P_{pos-1}] \end{Bmatrix}$$

$$ans = \max\{dp[N-1][balance]\}_{balance \geq 0}$$

While all of these would work, the final formulation is more efficient in terms of both run time and memory usage (and likely also cache performance), so we use that for the rest of our analysis.

There are two problems here. Firstly, the run time is $\Theta(N^2K)$ and, secondly, the memory usage is also $\Theta(N^2K)$. On the other hand, for subtask 2 we need $O(N^3)$ time and $O(N^2)$ memory, and for subtask 3 – $O(N^2)$ time and $O(N)$ memory.

There are two optimizations we can use. Either one of them is enough to get us subtask 2 and both are needed for subtask 3.

The first one divides the memory usage by N. Note that we only need to access the last three positions of the DP array. Therefore, instead of storing the entire DP array, we use a cyclic set of three buffers. Then, at each step we rotate them clockwise like so:

$$dpCurr, dpPrev, dpPrev2 = dpPrev2, dpCurr, dpPrev$$

The second optimization divides both the run time and the memory usage by K. The key observation is that while the possible values for the "balance" can range from $-NK$ to $+N$, we can never pay off any debt greater than N, as we can only pay off one unit of debt per node. Therefore, the useful values of "balance" range from $-N$ to $+N$. The final complexity is $\Theta(N^2)$ run time and $\Theta(N)$ memory usage.

Subtasks 4 and 5 for 50 p.

Here the gist of the solution is the same and the same sets of optimizations make the difference between subtasks 4 and 5. We only need to see what we need to do to handle a full cycle.

Let us start by seeing what goes wrong when we try to use the previous solution here. Almost everything is correct, except there is a single case we can't handle. The previous solution is based on the fact that we don't use the final node's outgoing edge. Therefore, we don't allow for the final node to be taken using its child's (node 0's) badge.

There is an easy enough fix for that: we add an additional binary dimension to the DP which denotes whether we chose to take the 0^{th} node or not. Then, when processing the final node, we allow it to take node 0's badge, if it wasn't used.

However, there is an alternative even simpler solution: we just run our previous solution twice: once starting from node 0 and once from its child. This works as we will never need to use both 0's outgoing edge and its child's outgoing edge (as doing that implies that we are taking the badge of a chosen node).

While both options work, the second one uses less memory and is likely to have better cache performance on top of being easier to code. Therefore, we will go with that for the rest of our analysis.

Finally, note that, while this solution is worth 50 points, it can easily be combined with the exponential solution for subtask 1, thus yielding us a total of 60 points. This goes for subtasks 2, 3 and 4 too.

Subtasks 6 and 7 for 100 p.

Now that we know how to handle a single cycle, we just need to combine the solutions for several cycles. Unfortunately, this is not as simple as adding up the final answers for each of the cycles, as we may borrow random badges from one cycle and use them up in another one. Therefore, we need to combine the full final DP states of each of the cycles. Note that for each cycle, we can take the point-wise maximum for each balance of all of the possible alternatives that lead to a final valid solution with that balance. So, for our solution that would be (here M_u is the length of node u's cycle and dp_u is the DP starting from u):

$$finalDp_u[balance] = \max\begin{Bmatrix} dp_u[M_u - 1][balance], \\ dp_{T_u}[M_u - 1][balance] \end{Bmatrix}$$

A merging better idea than the trivial exponential one is to keep a running total – the combined best cases for the cycles processed so far. Then, at each new cycle we merge its final DP state with the running total quadratically. If we do this with the slower ($\Theta(N^2K)$ per cycle) solution, we get a $\Theta(CN^2K^2)$ solution (C is the number of cycles), which is still not good enough. However, if we do this on top of the faster ($\Theta(N^2)$ per cycle) solution, we get a $\Theta(CN^2)$ solution, which is fast enough for subtask 6, as well as all of the ones below that. Therefore, we would get 80 points with this solution.

However, to get subtask 7 and the full 100 points, we need a pure $\Theta(N^2)$ solution. The way to get this is not merge the cycles at all. Instead, we keep a running total as before, but when we start processing a new cycle, we initialize its DP with the current running total (instead of negative infinities for all "balances" other than 0). This implicitly merges the new DP with the old ones, as it is using them as a base at no extra computational cost. Therefore, it works in $\Theta(N^2)$ time and gets 100 points. Note that using this type of implicit merging on top of the $\Theta(N^2K)$ per cycle solution yields a $\Theta(N^2K)$ solution for all cycles. This is good enough for subtasks 1, 2, 4 and 6, and thus yields a total of 70 points.

Problem Author: Emil Indzhev