Code Jam 2018 - Round 1B

Analysis: Transmutation

This is a very tricky problem with lots of pitfalls to consider.

Test set 1 (Visible)

Plain brute force is sufficient to pass this data set. There are many ways to approach it that would work, though. One intuitive approach would be to create one gram of lead (metal 1) as often as possible. Suppose `Create(i)` is a function that takes a metal i as parameter and returns true if it can create one gram of the i-th metal and false otherwise. Then, we can call `Create(1)` repeatedly until it returns false, and output the number of calls that returned true. An implementation of `Create(i)` works as follows:

- First, check if the remaining amount of the i-th metal is positive. If it is, we consume 1 gram of this metal and return true.
- Otherwise, let P and Q be the two metals that can be combined to create the i-th metal.
- We recursively call `Create(P)` and `Create(Q)`.
- If both of the function calls return true, we return true. Otherwise we return false.

However, there is slight problem with the above function. When we are not be able to create metal 1, this function will never end, it will keep trying to create the metals. Consider a simple example: there are 3 metals and for each of them the other two are the ingredients. Suppose, the amount of the metals are now 0 but we are trying to create lead. In such case, it will try to create the other two metals recursively, which will eventually try to create the lead again and so on. One way to fix it would be to check if we tried to create this i-th metal in the current call stack. If we already tried to create the i-th metal and while doing so we looked into its ingredient elements and again arrived at the i-th metal, that means it is not possible to create the i-th metal (and eventually the lead) anymore. Another way that is simpler to code is to limit the depth of the recursive call to **M**, because after **M** recursive calls we are guaranteed to repeat at least one metal.

The runtime of the above solution is not very large. The recursive call stack is up to **M** calls deep and at each level we call `Create` twice recursively. We may imagine it as a binary tree. The number of leaf nodes is at most $2^{\mathbf{M}}$ which makes the total less than $2^{\mathbf{M}+1}$. So, a `Create(1)` call takes $O(2^{\mathbf{M}})$ time. There may be at most 8 grams of each of these metals. So we will call `Create(1)` function about 8**M** times. For **M** = 8, this means the body of the function executes $8 \times 8 \times 2^8$, which is not too large.

Test set 2 (Hidden)

The approach above is of course too slow for the second test set. We can do a top-down approach as follows: we maintain a current "recipe" to create lead. The initial recipe is just use 1 gram of lead to create 1 gram of lead. The invariant is that the current recipe is always optimal, that is, any other way to create lead that can be done with the current supply is an expansion of the recipe. An expansion of a recipe is the result of zero or more applications of replacing 1 gram of a metal in the recipe by 1 gram of each of the two metals required to make it. This invariant is clearly true for the initial recipe.

As a first step, we make as much lead as we can with the current recipe, which we already mentioned is optimal. After doing that, the supply of one or more of the metals in the recipe is less than the recipe requirements of each. That means that any recipe that works is an

expansion of replacing 1 gram of each of those metals for its ingredients. We perform one of those replacements to get a new optimal recipe and repeat.

Notice that the total amount of grams of metal in the recipe only increases and the same number in the supply only decreases or stays the same (if we make no lead in a step). So, when the amount in the recipe surpasses the amount in the supply we can stop, since we won't be able to make another gram of lead.

Let S be the sum of all $\mathbf{G_i}$, that is, the amount of grams of metal in the initial supply. Each step above takes O(**M**) if we represent the recipe as a vector of size **M** with the required grams of each metal in the recipe. Checking how much lead we can do requires a single pass to find the limiting ingredient, and finding an ingredient that we need to replace requires another pass. Making the replacement of a single ingredient takes constant time. Since after each step the total amount of grams of metal in the recipe increases by at least 1, and the supply does not increase, the number of steps until the stopping condition is at most **S**. This makes the running time of this algorithm O(**M**S) which is enough to pass for $\mathbf{M} \leq 100$ and $S \leq 10000$.

Test set 3 (Hidden)

Since S can be up to 10^{11} for test set 3, we can't really use the approach above. Adding a lot of prunning to it to prevent really long cycles of replacements to happen can work, but it's hard to know exactly how much prunning is required unless we take a systematic approach. Fortunately, using binary search can simplify this a lot. First, we consider the simplified problem of deciding if it is possible to make L grams of lead, for an arbitrary L. If we can solve that efficiently, a simple binary search within the range [0, S] finished the problem, and multiplies the time complexity by a (relatively) small log S.

For a fixed L, we start by adjusting our supply by making the amount of lead $\mathbf{G_1}$ - L. That may leave us with lead debt instead of lead supply. We now iterate paying off debt until either we cannot or we have no debt left. If we cannot pay off some debt, then we cannot make L grams of lead. If we find ourselves with no debt left, we can make L grams of lead. While we iterate, we will adjust the supply **G** and the recipes **R**, that start as given in the input.

For each step of pay off, find an ingredient i such that $\mathbf{G_i} < 0$. If there are none, we paid off all debt and we are done. If there is, we go through the recipe to make metal i. If the recipe contains metal i itself, we can't pay off the debt and we are done. Otherwise, for each k grams required of metal j in the recipe, we do $\mathbf{G_j} := \mathbf{G_j} + k \times \mathbf{G_i}$ (remember $\mathbf{G_i}$ is negative). That is, we push the debt of metal i to requiring amounts of metals from its recipe. Finally we can set $\mathbf{G_i} := 0$. If we ever need i in the future, we know we will again need to go through its recipe, so we replace any k grams of i in the recipe of any ingredient by i's recipe multiplied by k. In this way, metal i will never be required in the future.

A step of payoff takes $O(\mathbf{M}^2)$ time: finding a metal in debt takes time linear on **M**, and so do adjusting all ingredients (remember we represent recipes by a vector of size **M** with the grams required for each metal). Replacing metal i in a single recipe takes time linear on **M** too, and we have to do it for each other of up to **M** metals, yielding $O(\mathbf{M}^2)$ time for the step.

During each step of payoff one metal disappears from the recipes. Since at most **M** - 1 metals can disappear (when there is only one metal i left, its recipe is guaranteed to contain metal i itself and we'll stop, since recipes only grow and thus are never empty) the total number of payoff steps is O(**M**). This makes the overall time to check an arbitrary L $O(\mathbf{M}^3)$, and the overall time for the entire algorithm $O(\mathbf{M}^3 \log S)$, which is fast enough to solve the hardest test set.