Code Jam 2018 - Round 1B

Analysis: Rounding Error

Test set 1

This test set can be solved using a complete search. We can try every possible partition of **N** voters among **N** languages. If two partitions differ only in the order of their languages, then we consider those partitions equivalent.

Therefore, we can consider a partition as an **N**-tuple $(x_1, x_2, ..., x_{\mathbf{N}})$, where $x_i \geq x_{i+1}$ and $\Sigma\, x_i = \mathbf{N}$.

Even with **N** = 25, there are no more than 2,000 different partitions. For each partition, we can use the following greedy algorithm to check whether the partition can be achieved by only adding voters: let us sort the $\mathbf{C}_i$ values in non-increasing order — that is, such that $\mathbf{C}_i \geq \mathbf{C}_{i+1}$. Then the partition can be achieved by only adding voters if and only if $\mathbf{x}_i \geq \mathbf{C}_i$ for all $1 \leq i \leq \mathbf{L}$. Among all such partitions, we can find the largest percentage sum, which is our answer.

Test set 2

For this test set, we can remove our assumption that a partition and **C** must be sorted non-increasingly. Therefore, we consider a partition of **N** voters to **N** languages as $(x_1, x_2, ..., x_{\mathbf{N}})$, where $\Sigma\, x_i = \mathbf{N}$.

To solve this test set, we can use dynamic programming (DP). We define a function f(a, b) as the following:

Among all partitions $(x_1, x_2, ..., x_a)$ such that $\Sigma\ (1 \leq i \leq a)\ x_i = b$ and $x_i \geq \mathbf{C}_i$ for all $1 \leq i \leq a$, what is the maximum $\Sigma\ (1 \leq i \leq a)$ round(x_i / **N** × 100) possible? If there is no satisfying partition, then f(a, b) = -∞. We can assume $\mathbf{C}_i = 0$ for $i > \mathbf{L}$.

We can first handle the base case of the function. We can easily compute f(1, b) since there is only at most one satisfying partition. Therefore, f(1, b) = round(b / **N** × 100) if $b \geq \mathbf{C}_1$, or -∞ otherwise.

The recurrence f(a, b) of this function can be computed by considering all possible values of x_a. Let i be the value of x_a. Therefore, x_a contributed round(i / **N** × 100) to the total percentage, and there are (b - i) votes left to be distributed among $x_1, x_2, ..., x_{a-1}$. Therefore, for a > 1, f(a, b) = max $(\mathbf{C}_a \leq i \leq b)$ (round(i / **N** × 100) + f(a - 1, b - i)).

Since we want to distribute **N** voters to $x_1, x_2, ..., x_{\mathbf{N}}$, the answer for the problem is f(**N**, **N**).

Function f has $O(\mathbf{N}^2)$ possible states and each state takes O(**N**) time to compute. Therefore, this solution runs in $O(\mathbf{N}^3)$ time.

Test set 3

We can solve this test set with a greedy strategy. For each language, we will either be rounding the percentage up or down. We get the maximum answer when as many of these as possible are rounded up.

Therefore, we can ignore any languages that are already being rounded up. Since there can be arbitrarily many languages, nothing ever forces us to disturb these languages by adding another vote—it's no worse to add that vote to some new language instead. We figure out how many more votes each language (including languages nobody has even mentioned yet) would need in order for it to be rounded up.

We greedily satisfy as many of these as possible, starting with the ones that take the fewest additional votes. This solution runs in O(**N** × log(**N**)) time.

Code Jam 2018 - Round 1B

Analysis: Mysterious Road Signs

Test Set 1 (Visible)

First, let's break down what the problem is asking us to find. We are asked to identify sets of contiguous signs, where each set is defined by four variables:

1. The index of the first sign in the set, *i* (inclusive)
2. The index of the last sign in the set, *j* (exclusive)
3. The destination for eastbound travelers, *M*
4. The destination for westbound travelers, *N*

In order for a set to be valid, each sign in the set must be truthful to eastbound travelers, westbound travelers, or both. Given the four variables above, we can evaluate a set of signs for validity in O(*j*–*i*) time. We can bound the number of possible values for *M* and *N* by **S** by taking the set of all westbound or eastbound destinations displayed by at least one sign. Since *i* and *j* are also bounded by **S**, we would need O($\mathbf{S}^5$) time for this solution, which is not sufficient for Test Set 1.

To improve our brute-force solution, we can be more clever with how we pick *M* and *N*. The first sign in a set tells us a lot of information: namely, it defines what either *M* or *N* must be (one of the two destinations on the sign, or else the sign wouldn't be able to be in the set). Suppose we fix *M* based on the first sign. Now, we can walk through the rest of the signs in the set until we find a sign whose eastbound destination is not *M*; use that sign's westbound destination as *N*. Continue walking until we reach a sign that does not share either *M* or *N*, in which case the set is invalid, or until we reach the end of the set, in which case the set is valid. We can perform the analogous process by fixing *N* based on the first sign. This “two-pass” set evaluation algorithm runs in O(**S**) time, since sets have size O(**S**).

Since there are O($\mathbf{S}^2$) sets of signs and we can evaluate each set for validity in O(**S**) time, we have a O($\mathbf{S}^3$) algorithm, which is fast enough for Test Set 1.

Test Set 2 (Hidden)

Clearly the cubic solution won't work on Test Set 2. It turns out that a quadratic solution coded in a fast language with a sufficient amount of pruning can pass Test Set 2. However, in this analysis, we will describe a O(**S** log **S**) solution followed by a linear-time O(**S**) solution, both of which comfortably finish in time when evaluating Test Set 2.

The O(**S** log **S**) solution is a classic application of divide and conquer. Cut the list of signs in half, except for a single sign at the center, which we will call the “midpoint” sign. Recursively apply the algorithm to the western half and the eastern half of signs. Then, use a modified version of the previously described two-pass algorithm to find the best set containing the midpoint: first, fix *M* to the midpoint's eastbound destination. Since we are looking for long segments, we can greedily add as many signs to the set as possible. Walk the list of signs both westward and eastward until reaching a sign on each end that does not share the same eastbound destination (*M* value) with the midpoint. Let N_1 equal the westbound destination of the western boundary (the first sign to the west that did not match *M*) and let N_2 equal the westbound destination of the eastern boundary (the firsts sign to the east that did not match *M*). Find M_1 and M_2 using an analogous procedure: return to the midpoint, walk westward and eastward until reaching signs that do not share a westbound destination (*N* value) with the

midpoint, and let M_1 and M_2 be the eastbound destination of the signs at the western boundary and eastern boundary, respectively. We now have four possible M/N pairs: (M, N_1), (M, N_2), (M_1, N), and (M_2, N). We can greedily walk east and west from the midpoint with each of the four pairs to find the longest set containing the midpoint. This “four-pass” step for D&C runs in linear time. Now that we have found the longest set(s) containing the midpoint as well as (recursively) the longest sets from the western and eastern halves of signs, we can combine these results to get the longest sets for the whole dataset. This yields an algorithm requiring T(**S**) operations to compute an input of size **S**, for a function T that satisfies T(**S**) = 2T(**S**/2) + O(**S**). Using the Master Theorem we get that T(**S**) = O(**S** log **S**).

The linear-time solution does a single pass, identifying all possible long segments. Start by walking forward from the first sign. Maintain two “candidates”, called the “M-candidate” and the “N-candidate”, with the following properties:

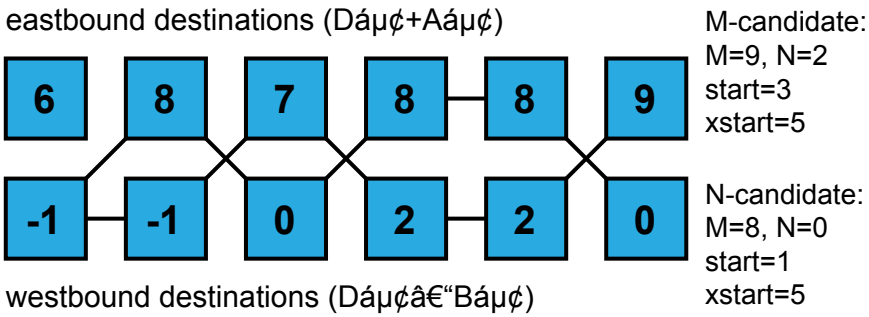
- Two destinations *M* and *N*.
- An index *start* corresponding to the westernmost sign in the contiguous segment of signs that includes the current sign and that satisfies *M* or *N*.
- An index *xstart* corresponding to the westernmost sign in the contiguous segment of signs that includes the current sign and whose eastbound destinations are all *M* (for the M-candidate) or whose westbound destinations are all *N* (for the N-candidate).

With these invariants, the set of signs starting at *start* and ending after the current index is guaranteed to be a valid set.

To maintain the invariants when reading a new sign, we can use the following procedure to create the new the M-candidate (the procedure for creating the new N-candidate is analogous):

- If new sign's eastbound destination equals previous sign's eastbound destination, copy the previous M-candidate to become the new M-candidate.
- If the new sign's eastbound destination equals the previous N-candidate's *M* value, copy the previous N-candidate to become the new M-candidate, and set *xstart* to the new sign's index.
- Otherwise, copy the previous N-candidate to become the new M-candidate, set *M* to the new sign's eastbound destination, set *start* to *xstart*, and set *xstart* to the new sign's index.

Consider the following illustration:



The illustration shows six zero-indexed signs and the destinations for those six signs. The signs are connected by lines that show which candidates are used when calculating the candidates for the next step. The candidates after reading the sixth sign (index 5) are shown at the end. On top we have the M-candidate with *M*=9 and *N*=2. The candidate starts at index 3, because the sign at index 2 does not have a compatible westbound or eastbound destination for that candidate. The illustration also demonstrates how *xstart*=5 is the start of the most recent string of westbound destination *M* matches, which in this case is only the current sign. The N-candidate goes all the way back to start index 1 with *M*=8 and *N*=0.

Since calculating the new candidates takes constant time, and we have to read each sign in sequence, this is an O(**S**) solution.

There are two more tips that can aid in solutions to this problem. First is that we don't ever need to remember $\mathbf{A_i}$, $\mathbf{B_i}$, and $\mathbf{D_i}$ directly; we only care about the westbound and eastbound destinations. You could therefore compute those destinations upon reading in the data and store them in a list of pairs. The second tip is that there is a lot of opportunity for pruning: if you maintain a global list of the best known sets of signs, you never need to look at any candidate sets that are smaller than the current best, allowing you to prune away a lot of segments that you no longer need to evaluate. All of the solutions described here except for the O(**S**) solution can take advantage of pruning.

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.