Kick Start 2018 - Round B

Analysis: Sherlock and the Bit Strings

Bit String: Analysis

The main challenge of finding the **P**-th lexicographically smallest valid string is to count the number of valid strings given a prefix requirement (i.e. the string must begin with a given prefix) and the **K** constraints given in the problem. The prefix requirement is necessary since we're going to fill the characters of the string one by one from the first character. Once we have solved the counting problem, the original problem becomes easier.

We can loop through each of the valid characters i in lexicographically increasing order (i.e. 0 and 1 in this problem) for the first position. Then we count the number of valid strings with prefix i. Let's say this value is X. If X ≥ **P**, we know that i is the right character for the first position. Otherwise, we continue the loop and subtract X from **P** until we find the right character for the first position. We keep doing this for each position until the last position.

To avoid an overflow when counting the number of valid strings, we can limit our computation to maxP, where maxP is the maximum value of **P** (i.e. 10^{18} in this problem). In other words, whenever we find that the number of valid strings would be more than maxP, we can set it to maxP.

Small dataset

Finding the number of valid strings in the Small dataset is easier since $\mathbf{A_i} = \mathbf{B_i}$ in each of the constraints. It is the same as saying that the $\mathbf{A_i}$-th character must be $\mathbf{C_i}$. Therefore, the set of possible characters for each position is independent of what character we choose for any other position. The number of valid strings is simply 2^Y, where Y is the number of positions without a constraint. For example, the number of valid strings without a given prefix requirement is $2^{\mathbf{N} - \mathbf{K}}$.

Large dataset

Finding the number of valid strings in the Large dataset is trickier since the set of possible characters for each position is no longer independent of what character we choose for any other position.

We can solve this dataset using a dynamic programming solution. Let us define a function `f(x, last)`, where `x` is a [1, **N**] integer and `last` is a 15-bit (15 is the maximum possible value of $\mathbf{B_j} - \mathbf{A_j}$) integer, is the number of ways of:

- assigning `0`s and `1`s to the bit string from the `x`-th position to the **N**-th position (in other words, assume that the bit string has been "filled" until the (`x` - 1)-th position),
- the last 15 characters (i.e. from (`x` - 15)-th position to the (`x` - 1)-th position) are represented in `last`, and
- the assignment has to satisfy all the constraints i where $\mathbf{B_i} \geq x$.

How do we calculate `f(x, last)`? We loop over i in the list [0, 1], the character that we consider putting in the `x`-th position. Together with `last`, we now know what the last 16 characters we have placed are. Since $\mathbf{B_j} - \mathbf{A_j} \leq 15$, and using the information of last 16 characters that we have placed, we can check all the constraints j where $\mathbf{B_j} = x$.

If at least one of these constraints is violated, then i is not a valid character to be placed in on the `x`-th position. Otherwise, placing in i on the `x`-th position contributes `f(x + 1, g(last, i))` to `f(x, last)`, where `g(last ,i)` is updating the `last` mask by ignoring the first bit of `last` (since we won't need the character from the (`x` - 15)-th position anymore) and appending an i bit to the end.

With the prefix requirement `s`, the number of valid strings is the value of `f(length(s) + 1, the last 15 characters of s)`. Since there are **N** possible values for `x`, 2^{15} possible values for `last`, and each computation of `f(x, y)` requires a loop for checking the constraints, this solution runs in $2^{15} \times O(\mathbf{N} + \mathbf{K})$ time. Even though 2^{15} is a constant multiplicative factor that does not affect the big-O time of the solution, it is relevant to the running time, particularly because 2^{15} is much greater than the largest possible value of **N** + **K**, so we include it here for clarity.