Code Jam 2021 - World Finals

Analysis: Divisible Divisions

View problem and solution walkthrough video

Test Set 1

In Test Set 1, $\mathbf{S}$ is reasonably small. This allows us to compute the answer for every prefix of the input using dynamic programming. We compute two values for every prefix ending at index i: A_i and B_i, which are the number of divisible divisions of this prefix whose last division is divisible by $\mathbf{D}$ (*Type A*) and the number of divisible divisions of this prefix whose last division is not divisible by $\mathbf{D}$ (*Type B*). The empty prefix has corresponding values $A_0 = 1$ and $B_0 = 0$. The solution to the problem is $A_{\text{length}(\mathbf{S})} + B_{\text{length}(\mathbf{S})}$.

Let $\mathbf{S}[i..j]$ be the integer represented by the digits in $\mathbf{S}$ from index i to index j, inclusive. If $i > j$, then this is an empty substring and the corresponding value is 0 (this only happens with $j = 0$ below). To compute A_k, we will iterate over all possible last divisions ($\mathbf{S}[1..k], \mathbf{S}[2..k], \ldots, \mathbf{S}[(k-1)..k], \mathbf{S}[k..k]$). For each one of these substrings (say $\mathbf{S}[i..k]$), we check if it is divisible by $\mathbf{D}$, and if it is, we can append this division to any divisible division that ends at index $i-1$. Since the division we constructed is divisible by $\mathbf{D}$, it does not matter if the divisible division we are appending to ends in a division that is divisible by $\mathbf{D}$. The formulas below use modular arithmetic notation. Thus,

$$A_k = \sum_{\substack{1 \le i \le k \\ \mathbf{S}[i..k] \equiv 0 \pmod{\mathbf{D}}}} (A_{i-1} + B_{i-1})\,.$$

For convenience below, we slightly rewrite this equation as:

$$A_k = \sum_{\substack{0 \le i \le k-1 \\ \mathbf{S}[(i+1)..k] \equiv 0 \pmod{\mathbf{D}}}} (A_i + B_i)\,.$$

We can compute B_k similarly. We iterate over all possible last divisions. For each one of these substrings that is not divisible by $\mathbf{D}$, we can append it to any divisible division that <u>does</u> end with a division that is divisible by $\mathbf{D}$. Thus,

$$B_k = \sum_{\substack{0 \le i \le k-1 \\ \mathbf{S}[(i+1)..k] \not\equiv 0 \pmod{\mathbf{D}}}} A_i.$$

Since |**S**| is small, we can simply check all possible values. Some care is needed when determining whether $\mathbf{S}[i..k]$ is divisible by $\mathbf{D}$. Using the fact that $\mathbf{S}[i..k] = 10^{k-i} \cdot \mathbf{S}[i] + \mathbf{S}[(i+1)..k]$, we can avoid fully recomputing the value of $\mathbf{S}[i..k]$ by keeping this rolling value (as well as maintaining the current power of 10).

For each k, computing A_k and B_k requires a linear sweep through all smaller indices, so this solution takes $O(|\mathbf{S}|^2)$ time.

Test Set 2

When $\mathbf{D}$ and 10 are relatively prime

In Test Set 2, $\mathbf{S}$ is too large to use the dynamic programming solution explained above. However, we will make use of the same foundation for our solution: for each index, compute A_k and B_k.

Rather than sweeping through all smaller indices to check which prefixes are divisible by $\mathbf{D}$ and which are not, we will instead use the following observation:
$\mathbf{S}[(i+1)..k] = \mathbf{S}[1..k] - \mathbf{S}[1..i] \cdot 10^{k-i}$. To find all i such that $\mathbf{S}[(i+1)..k] \equiv 0 \pmod{\mathbf{D}}$ (which is needed in the formula above for both A_k and B_k), we instead search for all $0 \le i \le k-1$ such that $\mathbf{S}[1..i] \cdot 10^{k-i} \equiv \mathbf{S}[1..k] \pmod{\mathbf{D}}$. This allows us to group all terms in the summation by the value of $v \equiv S[1..i] \pmod{\mathbf{D}}$:

$$\mathcal{A}_v^{(k)} = \sum_{\substack{0 \le i \le k-1 \\ \mathbf{S}[1..i] \cdot 10^{k-i} \equiv v \pmod{\mathbf{D}}}} A_i,$$

and similar for $\mathcal{B}_v^{(k)}$.

With this framework, we can re-write our formula above for A_k and B_k:

$$A_k = \left(\mathcal{A}_{\mathbf{S}[1..k]}^{(k)}\right) + \left(\mathcal{B}_{\mathbf{S}[1..k]}^{(k)}\right) \qquad \text{and} \qquad B_k = \sum_{v \ne \mathbf{S}[1..k]} \mathcal{A}_i^{(k)} = \left(\sum_{0 \le v < \mathbf{D}} \mathcal{A}_v^{(k)}\right) - \mathcal{A}_{\mathbf{S}[1..k]}^{(k)}$$

The only piece of the puzzle left is determining how to compute $\mathcal{A}_v^{(k)}$ and $\mathcal{B}_v^{(k)}$ quickly. Intuitively, to move from $\mathcal{A}_v^{(k)}$ to $\mathcal{A}_v^{(k+1)}$, we must do two things: (1) multiply every index v by 10 modulo $\mathbf{D}$, then (2) apply our knowledge of A_k and B_k. In the equations below, we need to make use of the multiplicative inverse of 10 modulo $\mathbf{D}$, 10^{-1} (this is why we need $\mathbf{D}$ and 10 to be relatively prime). Mathematically, we can write $\mathcal{A}_v^{(k+1)}$ as follows (proofs of these are at the very bottom): if $\mathbf{S}[1..k] \equiv v \pmod{\mathbf{D}}$, then

$$\mathcal{A}_v^{(k+1)} = \mathcal{A}_{10^{-1}v}^{(k)} + A_k$$

and if $\mathbf{S}[1..k] \not\equiv v \pmod{\mathbf{D}}$, then

$$\mathcal{A}_v^{(k+1)} = \mathcal{A}_{10^{-1}v}^{(k)}.$$

If we wish to store $\mathcal{A}_v^{(k+1)}$ as an array, we could naively loop through every index v of $\mathcal{A}_v^{(k)}$ (but that would be much too slow). Instead, we do the multiplication implicitly. If an index is v in $\mathcal{A}^{(k)}$, then it is at index $10v \pmod{\mathbf{D}}$ in $\mathcal{A}^{(k+1)}$. This means that after applying (1) above, to compute the value of $\mathcal{A}_v^{(k+1)}$, we can instead examine $\mathcal{A}_{10^{-1} \cdot v}^{(k)}$. We can recursively apply this logic and store just one array: $\mathcal{A}^{(1)}$ and look at the appropriate index:

$$\mathcal{A}_i^{(k+1)} = \mathcal{A}_{10^{-k} i}^{(1)}.$$

In the case where $\mathbf{S}[1..k] \equiv v \pmod{\mathbf{D}}$, we accomplish (2) similarly by increasing $\mathcal{A}_{10^{-k} \cdot \mathbf{S}[1..k]}^{(1)}$ by A_k (and similar for $\mathcal{B}$).

In total, we can keep track of all of these operations in $O(|\mathbf{S}|)$ time and $O(\mathbf{D})$ memory. We do need one extra variable to store $\left(\sum_{0 \le v < \mathbf{D}} \mathcal{A}_v^{(k)}\right)$ for the computation of B_k, but this is easy to maintain in $O(1)$ time per k.

Chinese Remainder Theorem to the rescue!

The above algorithm works because $\mathbf{D}$ and 10 were assumed to be relatively prime. This was needed for 10^{-1} to exist in all cases. But what do we do when they are not? We write $\mathbf{D} = 2^\ell 5^m n$, where $\gcd(n, 10) = 1$. Instead of checking that $\mathbf{S}[i..k] \equiv 0 \pmod{\mathbf{D}}$, we will instead break it up into three separate (simultaneous) checks: $\mathbf{S}[i..k] \equiv 0 \pmod{2^\ell}$, $\mathbf{S}[i..k] \equiv 0 \pmod{5^m}$, and $\mathbf{S}[i..k] \equiv 0 \pmod{n}$. By the Chinese Remainder Theorem, all three of these are true if and only if $\mathbf{S}[i..k] \equiv 0 \pmod{\mathbf{D}}$.

Recall that we are searching for all i such that $\mathbf{S}[1..i] \cdot 10^{k-i} \equiv \mathbf{S}[1..k] \pmod{\mathbf{D}}$. The key observation needed is that if $k - i \geq \ell$, then $\mathbf{S}[1..i] \cdot 10^{k-i} \equiv 0 \pmod{2^\ell}$. Similarly, if $k - i \geq m$, then $\mathbf{S}[1..i] \cdot 10^{k-i} \equiv 0 \pmod{5^m}$. This means that a substring of $\mathbf{S}$ (say $\mathbf{S}$[i..k]) that is longer than $\max(\ell, m)$ can only contribute to $\mathcal{A}$ if $\mathbf{S}[1..k] \equiv 0 \pmod{2^\ell}$ and $\mathbf{S}[1..k] \equiv 0 \pmod{5^m}$.

This leads us to our solution. For each k, we will compute A_k and B_k by breaking into two cases: the "small" substrings and "large" substrings. For "small" substrings (that is, substrings of length at most $\max(\ell, m)$), we use our algorithm from Test Set 1, looping through all small substrings naively.

For the "large" substrings, we know that if $\mathbf{S}[1..k] \not\equiv 0 \pmod{2^\ell}$ or $\mathbf{S}[1..k] \not\equiv 0 \pmod{5^m}$, then no large substrings are divisible by $\mathbf{D}$. So the large part of $A_k = 0$ and the large part of $B_k = \sum_{0 \leq v < \mathbf{D}} \mathcal{A}_v^{(k)}$. If, however, $\mathbf{S}[1..k] \equiv 0 \pmod{2^\ell}$ and $\mathbf{S}[1..k] \equiv 0 \pmod{5^m}$, then we can use the technique described above (with n instead of $\mathbf{D}$). One small modification is needed: do not add in our knowledge of A_k and B_k into $\mathcal{A}$ or $\mathcal{B}$ until they are out of range of the "small" substrings or else the "small" substrings will be counted multiple times.

Computing the "large" substrings takes linear time (as explained in the section above). To compute the "small" substrings, we must naively loop over $\max(\ell, m)$ elements. Note that $\max(\ell, m) \leq \log_2 \mathbf{D}$. This means that, in total, we do $O(|\mathbf{S}| \log \mathbf{D})$ operations.

Proofs

If $\mathbf{S}[1..k] \equiv v \pmod{\mathbf{D}}$, then:

$$\begin{aligned}
\mathcal{A}_v^{(k+1)} &= \sum_{\substack{0 \leq i \leq k \\ \mathbf{S}[1..i]\cdot 10^{k+1-i} \equiv v \pmod{\mathbf{D}}}} A_i \\
&= \left(\sum_{\substack{0 \leq i \leq k-1 \\ \mathbf{S}[1..i]\cdot 10^{k+1-i} \equiv v \pmod{\mathbf{D}}}} A_i \right) + A_k \\
&= \left(\sum_{\substack{0 \leq i \leq k-1 \\ \mathbf{S}[1..i]\cdot 10^{k-i} \equiv 10^{-1} v \pmod{\mathbf{D}}}} A_i \right) + A_k \\
&= \mathcal{A}_{10^{-1}v}^{(k)} + A_k
\end{aligned}$$

Otherwise, if $\mathbf{S}[1..k] \not\equiv v \pmod{\mathbf{D}}$, then:

$$\begin{aligned}
\mathcal{A}_v^{(k+1)} &= \sum_{\substack{0 \le i \le k \\ \mathbf{S}[1..i]\cdot 10^{k+1-i} \equiv v \pmod{\mathbf{D}}}} A_i \\
&= \left(\sum_{\substack{0 \le i \le k-1 \\ \mathbf{S}[1..i]\cdot 10^{k+1-i} \equiv v \pmod{\mathbf{D}}}} A_i \right) \\
&= \left(\sum_{\substack{0 \le i \le k-1 \\ \mathbf{S}[1..i]\cdot 10^{k-i} \equiv 10^{-1} v \pmod{\mathbf{D}}}} A_i \right) \\
&= \mathcal{A}_{10^{-1}v}^{(k)}
\end{aligned}$$