Code Jam 2010 - Round 3

Analysis: Different Sum

The small input

The solution for the small input of this problem was quite straightforward. One could iterate over all possible ways to partition **N** into a sum of positive integers, and verify that each column has distinct digits.

Since there are 190569292 partitions of 100 into a sum of positive integers, this algorithm might not run fast enough. However, we can optimize it with an easy observation: the summands must be distinct. That brings the total number of partitions of 100 down to just 444793, which is small enough for our needs.

But if you want to cut the search space down even further, you can use *backtracking*. This is a general technique that works as follows in this problem: as you're generating the partition, you can check if there's a column that has two equal digits after adding each number, not just in the end. That way, many bad partitions get filtered out early and you have even less possibilities to check.

The large input

In order to approach the large input, we need to rotate ourselves 90 degrees. In the above solution, we've generated our cryptarithm from top to bottom. Now, we will generate it from right to left.

First, we check all possibilities for the digits in the rightmost (least significant) column such that the last digit of their sum matches the required one. Then, we continue with the digits for the next-to-rightmost column, and so on.

Suppose we have already filled a few rightmost columns. We can note that the things that are relevant for us now is the value **V** of carry from the already filled columns to the next one, the amount **K** of summands in the column that was just filled, and the boolean flag **F** indicating whether there has been a zero in the column that was just filled (this flag is important since it affects whether we can terminate the corresponding number now). When we know the values of **V**, **K** and **F**, the actual digits in the already filled columns don't affect the further execution of the algorithm.

This observation logically leads us to the following *Dynamic Programming* solution: let's calculate **Count**[**i**, **V**, **K**, **F**] which is defined as the number of ways to place the digits in the last **i** columns in such a way that the sum in those columns matches **N**, there's a carry of **V**, the number of summands that have at least **i** digits is **K**, and **F** is 1 when there's a summand that starts with zero, 0 otherwise.

In order to calculate **Count**[**i**+1,...] given **Count**[**i**,...], we need to consider all possible ways to place up to **K** digits in the **i**+1-th rightmost column. **K** is up to **B** (since all digits in one column are different, the number of summands doesn't exceed the number of different digits), which can be up to 100 in the large input. From the first glance, this gives us at least 100! (factorial of 100) possibilities, rendering our idea still useless.

But now's when another Dynamic Programming idea comes into play! One can notice that we don't need to know exactly all digits of the **i**+1-th column. The important thing for us is the amount of those digits, the sum of those digits, and whether one of them is zero. When we know those, we can multiply our answer by an appropriate number (which will be a product of

binomial coefficients and factorials) to account for various ways to attach those digits to the already formed numbers in the first **i** columns.

So we run a separate Dynamic Programming that calculates **Count2**[**K**, **S**, **F**] which is defined as the number of ways to place **K** distinct digits in a column such that their sum is **S** and **F** denotes whether one of them is zero. **K** is up to **B**, **S** is O(**B**^{2}), meaning we get O(**B**^{3}) states, which is small enough.

The main Dynamic Programming has O(**B**^{2}***number_of_digits**) states, and using the **Count2** table each state can be processed in O(**B**^{2}) by looking at the number of digits in the **i**+1-th column and the carry to the **i**+2-th column (the required sum in the **i**+1-th column is uniquely determined by the carry to it, the carry from it, and the corresponding digit of **N**). The total runtime of this solution is thus O(**B**^{4}***number_of_digits**).