Code Jam 2020 - Virtual World Finals

Analysis: Hexacoin Jam

The main task in this problem is to count how many ways there are to pick a permutation and a pair of numbers from the list such that their (possibly truncated) sum falls within the given interval. This is the numerator of a fraction representing the desired probability. The number of total picks (our denominator) is simply 16! × **N** × (**N** - 1) / 2. Then, we can use a known algorithm (like dividing numerator and denominator by the greatest common divisor) to reduce the fraction to the desired form. In the rest of the analysis, we focus only on calculating the non-reduced numerator.

One common denominator among all solutions for all test sets is that the actual values of the digits in the list do not matter. Once we fix a pair of numbers X and Y from the list, only the "digit structure" of the pair (X, Y) matters. The digit structure of a pair of numbers is a unique way to see their coupling as digits. We define it as the lexicographically smallest pair (P[X], P[Y]) across all possible digit permutations P, where P[X] is the result of replacing each digit of X with the value assigned to it by P. Notice that digit structures have length 2**D**.

The total number of digit structures grows rapidly when **D** increases, but depends only on **D**, which has small limits. The total number is 15 for **D**=2, 203 for **D**=3, 4140 for **D**=4, and 115975 for **D**=5.

Test Set 1

In Test Set 1, the number of different digit structures is really small. In addition, for a fixed structure, we only care about the values assigned by the chosen permutation to up to 2**D** digits. For each digit structure with d unique digits, we can compute the value of the truncated sum for a valid assignment of those d digits, noting that there are (16 - d)! ways to generate each assignment.

The number of valid assignments can be somewhat large at 16! / 10! or about 6 million for 6 different digits, but there is only one digit structure within Test Set 1 that has that many different digits. There are a handful that have 5, for which there are around half a million assignments each, and most structures have 4 or fewer different digits, which have fewer than 50 thousand different assignments each. Moreover, this computation only depends on **D** and not on the rest of the input, so we have to do it only once for each possible structure for **D**=2 and **D**=3.

Given the precomputation above, we can iterate over the pairs of integers from the list, compute their digit structures, and use two binary searches to see how many of the numbers in the list are in range. Then, we compute the number of different digits in the structure d and multiply the total by (16 - d)!, which gives us the number of sums in range that can be produced for the given pair. Summing that result over all possible pairs gives us the answer we need.

Test Set 2

The precomputation above can be slow in Test Set 2. Not only do we have 4140 additional digit structures to process, but most importantly, a few of those have 7 and 8 unique digits. Each additional digit means an order of magnitude extra possible assignments. There are several ways to handle this, and we need only a few of the tricks below to make it work.

The first issue is that the lists we need to store are now too long to fit in memory. Since there are only up to 16^4 different results, many of those results would be repeated, so we can compress them based on those repetitions. This is still tough to pull off, so the best thing to do is to just not

store the full list. We know we only care about how many items on the list are in range for up to **T** different ranges. So, we can read all cases before starting the computation, split the full range of sum results at every **A** and **B** that we read into up to 2**T**+1 minimal ranges, and then compress together all numbers that are within the same range. This definitely fits in memory.

We can also choose to not memoize between different test cases, and treat them one at a time. If we do that, we have a fixed **A** and **B**, so there is no need for lists, just a counter. We can either treat each pair of numbers individually as well (speeding up the process of assignments — see below) or try to memoize digit structures if any are repeated within the same test case. It is possible that almost every pair has a different digit structure, of course, but there are few digit structures with the maximum number of unique digits. This means the memoization reduces the total runtime in what was our previously worst case, and the new worst case (all different structures) is not as bad because many of those structures will have fewer different digits.

We can reduce the number of digit structures further by realizing that digit structures like (011, 022) and (012, 021) are equivalent, in the sense that for any assignment, their sum is the same: both are 11 × (P[2] + P[3]) + 100 × P[0], where 11 and 100 are in base 16. This only works in conjunction with some form of memoization.

Once we have either a range [**A**, **B**] or a small list of ranges fixed at the moment of processing the assignments, we can use it to do some pruning. Suppose we assign values to the digits in most significant positions first. When we have done a partial assignment, we can compute or estimate a range of possible values that the sum may have when we finish. As more highly significant digits get assigned, that range shrinks. If that range is completely inside our target range (or one of our target ranges) we can stop and know that all further assignments work, counting them using a simple multiplication. If the range is completely outside of the target range (or all target ranges), we can also stop and count nothing further.

As mentioned above, we need only some of the optimizations above to manage to pass Test Set 2. Of course, the more of them we find and implement, the more confident we can be about the speed of our solution.

Test Set 3

To simplify the problem, we can use a common technique when dealing with counting numbers in closed intervals [A, B]. We write a function f(U) that only calculates the value for closed intervals [0, U-1]. Then, the result for an interval [A, B] is f(B + 1) - f(A). In this case, we can use this to take care of the overflow as well, by simply ignoring the overflow and counting the number of hits in the interval [A, B] plus the number of hits in the interval [$16^{\mathbf{D}}$ + A, $16^{\mathbf{D}}$ + B], which are the only possible sums whose truncation would yield a result in [A, B]. After we write our function f, this translates to a result of f(B + 1) + f($16^{\mathbf{D}}$ + B + 1) - f(A) - f($16^{\mathbf{D}}$ + A). We focus now on calculating f(U), that is, the number of picks of a permutation and a pair of numbers that yield a (non-truncated) sum of strictly less than U.

Both the number of possible values a sum can have and the possible number of pairs are small (for computers). We can use an asymmetric "meet in the middle" approach to take advantage of both those facts. We do this in a similar way as what we did for Test Set 1 and possibly for Test Set 2, by keeping a count on each digit structure, and then iterating over the pairs of numbers from the list to see how much we have to use each of those counts.

First, let's consider all the ways to add up to something less than U. Let us fix the first number to have a value of x, so the second number can have any value less than U-x. A number y is less than U-x if and only if it has a smaller digit than U-x at the first position at which they differ. We can represent this set of ys by saying they are all the ys that start with the first i digits of U-x, then continue with a digit d smaller than the (i+1)-th digit of U-x, and any remaining digits can be any digit.

For example, if U=2345 and x=1122, then U-x=1223 and y can be of the form 0***, 10**, 11**, 120*, 121*, 1220, 1221, 1222, where * represents any digit.

For each pair of an x and a prefix for y, we can represent all the pairs of numbers from the list that match by matching that with a digit structure that allows for *s at the end. In this way, a pair of numbers from the list X and Y can be mapped to x and y by a permutation if they have the same digit structure, disregarding the actual digit values. To represent this, we normalize the digit structure as we did before: no digit appears for the first time before another smaller digit. Therefore, a structure like x=1122 and y=10** is represented as x=0011 and y=02**. As before, the number of permutations that can match a pair of numbers to this digit structure is (16 - d)!, where d is the number of unique digits that appear in the structure.

Notice that different values of x can yield the same structure. For example, x=1133 yields U-x=1212, and for y=10** the structure is the same as for x=1122 and y=10**.

For each digit structure of 2**D** total digits that have between 0 and **D**-1 asterisks at the right end, we count the number of permutations that make its parts add up to something less than U.

We now process the pairs of numbers from the list. For each pair, we build its normalized digit structure as before, and add the count for that digit structure to our running total. We also add the count of the structures that result in replacing up to **D**-1 rightmost digits with *s.

We can express the complexity of this algorithm in terms of the base B, the number of digits of each number **D**, and **N**, the size of the list of numbers. The first stage iterates through up to $O(B^{\mathbf{D}})$ possible values for x, and for each one, considers up to O(B×**D**) digit structures for y. If factorials up to B are precomputed, and we store the results in a hash table or an array with a clever hashing for digit structures that keeps hashes unique and small, this requires only constant time per pair, so $O(B^{\mathbf{D}+1}\times\mathbf{D})$ time overall. The second stage requires processing up to O(**D**) digit structures per pair, so it takes $O(\mathbf{N}^2\times\mathbf{D})$ time in total. The sum of both stages gives us the overall time to compute f, which is $O(B^{\mathbf{D}+1}\times\mathbf{D} + \mathbf{N}^2\times\mathbf{D})$. Since we need only a constant number of calls to f, that is also the overall time complexity of our algorithm.

Another solution is to to use all of our Test Set 2 tricks in an efficient way. Consider especially the last one: when considering the pair of i-th most significant digits, only the pairs whose sum is close to either that value at **A** or that value at **B** produce a range of possible sums for the pair that is neither fully inside [**A**, **B**] nor fully outside of it. That means that for the i-th digits, there are only a linear number of pairs of digits that can be assigned at that position that would make the process not stop immediately, instead of a quadratic number. This shrinks the number of assignments to approximately the square root of its previous value, behaving more as exponential on **D** than as an exponential on 2**D**. This gives us a time complexity comparable to that of the other solution presented above. While the overall time complexity is still higher, these two solutions can perform pretty similarly for the limits we have: the bound on the number of assignments is actually smaller than $B^{\mathbf{D}}$ because it behaves more like B! / (B - **D**)!. Those savings, plus those from not needing extra **D** terms, make up for the fact that the backtracking solution's time complexity has a term with behavior similar to $B^{\mathbf{D}}$ and a term with behavior similar to $\mathbf{N}^2$ multiplied together instead of added.

Notice that the basic insights of both solutions are closely related. The pruning provides such a large speedup for the same reason that we can use the "meet in the middle" approach based on digit structures.