Code Jam 2017 - World Finals

Analysis: Operation

Operation: Analysis

Small dataset

Let us consider the Small dataset first. The number of operations is large enough that trying all possible orderings would time out, but a typical trick might work: turn that **N**! into a $2^{\mathbf{N}}$ with dynamic programming over subsets of cards. That is, try to reduce all possible orderings of a subset of the cards to only few possible results that are worth exploring further. The first option to try is to define a function f(C), for a subset C of the cards, as the best possible result of applying cards in C to the starting value **S**. It seems that defining f(C) as the maximum over all c in C of applying c to f(C - c) could be reasonable (with f(∅) being **S**). However, it doesn't quite work in general. For instance, suppose c is `* -1`. We are getting the maximum possible result out of f(C - c), only to flip the sign right after. It would have been better to get the minimum possible result for f(C - c) instead. Of course, if c is `* 2` instead, getting the maximum for f(C - c) seems like a good idea. It turns out that the best option is always either the minimum or the maximum for f(C - c), which we prove below. Therefore, let g(C, maximum) be the maximum possible result of applying cards in C to **S**, and g(C, minimum) be the minimum among those results. We can define g(C, m) recursively as the "m" (i.e., the maximum or the minimum depending on m) over each c and each m' in {minimum, maximum} of applying c to g(C - c, m'), with g(∅) = **S**. This definition of g formalizes our intuition that only the minimum and maximum possible values are needed from each subset. We prove its correctness below. The result is then g(C, maximum) for C = the entire input set. The function has a domain of size $2^{\mathbf{N}} \times 2$, and calculating each value involves an iteration over at most $2 \times \mathbf{N}$ possibilities, yielding $O(2^{\mathbf{N}} \times \mathbf{N})$ operations in total after memoizing the recursion. Of course, those operations are over large-ish integers. The number of digits of those integers is bounded by O(**N**D) where D is the number of digits of the input operands. That means the time complexity of each operation, which operates on a large integer and an input integer with up to D digits, is bounded by $O(\mathbf{N}D^2)$, which makes the overall running time of this algorithm $O(2^{\mathbf{N}} \times \mathbf{N}^2 \times D^2)$.

We can prove the definition of g is correct by complete induction. If C is the empty set, then g is correct immediately by definition. Otherwise, assume g is correct for all m' and all sets C' smaller than C, and let us prove that g(C, m) is correct. Let c be the last card used in an ordering of C that gives the "m" result when applied to **S**. If c is +v or -v, we can commute the operator "m" with the application of c. That is: let T be the result of applying all of the other operations in the optimal order. Then we know that T + v or T - v is "m" over the operations, so if the value of T is not the optimal g(C - c, m), then there is some other ordering that yields g(C - c, m) + v or g(C - c, m) - v, which is better. The same is true for c being a multiplication or division by a non-negative, since those also commute with maximum and minimum. If c is a multiplication or division by a negative, then it can commute with a maximum or minimum operator, but the operator is reversed, that is, max turns into min, and vice versa. Since we try both maximum and minimum in the definition of g, that poses no problem. Notice that this proof also shows that we do not even need to try both options for m'; we only need to check the one that actually works. Trying both is simpler, though, and it doesn't impact the running time in a significant way.

Large dataset

Of course, an exponential running time would not work for the Large dataset, so we need to reason further. As a first simplification, assume all additions and subtractions have positive operands by removing those with a zero operand, and flipping both the sign and the operand of those with a negative operand. This leaves an input with the same answer as the original one.

Suppose all cards are already ordered forming an expression E. We can distribute to "move" all additions and subtractions to the right end creating a new expression E' that contains multiplications and divisions first, and additions and subtractions later. In order to make the value of E the same as the value of E', we change the added or subtracted value on each term. The value of a given addition or subtraction card is going to be multiplied/divided by all multiplications/divisions that are to its right in E. For instance, if E = (((0 + 1) × 4) - 6) / 2, then E' = ((0 × 4) / 2) + 2 - 3. Notice "+ 1" turned into "+ 2" because it is multiplied by 4 and divided by 2. "- 6" turned into "- 3" because it is only divided by 2 (the multiplication in E does not affect it).

If we consider an initial fixed card "+**S**", we can even move that one to the end and always start with a 0, making multiplications and divisions in E effectively not needed in E'. The final result is then the sum over all additions minus the sum over all subtractions of the adjusted values (in the example above 4 is the adjusted value of the only addition and 3 is the adjusted value of the only subtraction). Notice that this shows the value of **S** always impacts the final result in the same way regardless of the order: it adds **S** times the product of all multiplications and divided by all divisions to the result.

Consider a fixed order for multiplications and divisions, and insert the additions and subtractions. As explained in the previous paragraph, inserting operation Z at a given position implies that Z will get multiplied by all multiplications that are to its right, and divided by all divisions that are to its right. Effectively, each position has a fraction F such that operations inserted there get multiplied by F. Given the view of the final result above, it follows that it's always best to insert additions at a position where F is maximal, and subtractions at a position where F is minimal. Even though there could be multiple places with the same value of F, this shows that it is never suboptimal to insert all additions in the same place A, and all subtractions in the same place B. Since additions and subtractions commute, this further shows that it is equivalent to have an input with a single addition and a single subtraction whose operand is the sum of the operands of the corresponding operation (after adjusting the signs to make them all positive).

Given the simplification, we can reverse the point of view. Consider the addition and subtraction fixed and insert multiplications and divisions. Since multiplication and division commute, we just need to decide between 3 places: a) before both addition and subtraction, b) in between, c) after both. What we insert in a) will multiply/divide only **S** in the final result, what we insert in b) will multiply/divide **S** and only additions or only subtractions depending on which is earlier, and what we insert in c) will multiply/divide everything. If we fix the positions of multiplications and divisions with negative operands, we can greedily place multiplications and divisions with a positive operand: we place multiplications to apply to the greatest of the 3 values mentioned above (after applying the fixed multiplications and divisions by a negative) and divisions to apply to the lesser of the 3 (after doing the same). This shows that it is never suboptimal to place all multiplications by a positive together in one place, and all divisions by a positive in one place.

To further simplify, divisions can't have a zero operand, but multiplications can. Notice that a "* 0" will nullify the entire thing, so only the placement of the rightmost "* 0" matters, so we can simplify them all into a single card (or no card if there is no "* 0" in the input). This leaves only multiplications and divisions by a negative. If a pair of multiplications by a negative are in the same place a), b) or c) as above, they multiply as a positive, so it is always better to move the pair to the optimal place to put multiplications, as long as we have pairs. If there is an even number of multiplications by a negative in a suboptimal place, then all of them get moved by this. If their number is odd, all but one are moved. Leaving behind the one with the smallest absolute value is optimal. A similar argument applies to divisions by a negative, although the optimal place to move to may of course be different than the optimal place to move

multiplications. This shows that we can group multiplications and divisions by a negative similarly to what we did with all other operations, but leaving out the two smallest absolute values of each type, as they may be needed in suboptimal places to perform a sign change (there are 3 places out of which 1 is optimal, leaving 2 suboptimal places).

After all the groupings, we are left with at most 11 cards: 1 addition, 1 subtraction, 1 multiplication by 0, 1 multiplication by a positive, 1 division by a positive, 3 multiplications by negatives and 3 divisions by negatives. This can be refined further, but there's no need for it. With just 11 cards, we can apply the Small solution and finish the problem. There are also other ways of reasoning among similar lines to get the number of cards low enough. Another possibility that doesn't require any dynamic programming is to notice that we can brute-force the order of the addition and subtraction (two options), and then brute-force which place a), b) or c) each multiplication and division (up to 8 cards after all simplifications) should go into. This requires exploring only 2×3^8 combinations in total, and exploring each one is a relatively fast process requiring only $O(\mathbf{N}^2)$ time (11 operations of quadratic time each, since the operands in the simplified cards can have linear size).

It is possible to reduce the set of cards further with more thought, and it's also possible to follow other lines of reasoning that will lead you to slightly higher card totals that are small enough to make the dynamic programming solution work.