Daily Coding Problem #308

Problem

This problem was asked by Quantcast.

You are presented with an array representing a Boolean expression. The elements are of two kinds:

- `T` and `F`, representing the values `True` and `False`.
- `&`, `|`, and `^`, representing the bitwise operators for `AND`, `OR`, and `XOR`.

Determine the number of ways to group the array elements using parentheses so that the entire expression evaluates to `True`.

For example, suppose the input is `['F', '|', 'T', '&', 'T']`. In this case, there are two acceptable groupings: `(F | T) & T` and `F | (T & T)`.

Solution

Though it may not seem so at first, this is actually a dynamic programming problem.

Let us review what that means. First, it should be possible to break our problem down into overlapping subproblems. The simplest subproblems here are if we restrict our evaluation to just the operands in our input. For each of these, there is either one way to evaluate this

expression truthfully (if the operand is `True`), or zero (if the operand is `False`).

As we look at larger and larger input sections, we see that each one can be considered a problem as well. For example, in the example above, we can try to figure out how many ways `F | T` can be `True`, and then `T & T`, and finally the entire list.

The second property required is that of optimal substructure; that is, we should be able to use the results from smaller problems to build up solutions to the larger ones. With some cleverness, this can be achieved as well.

We can first initialize two matrices, `T` and `F`. The value contained in `T[i][j]` will represent the number of ways the subsection of the input from operand `i` to operand `j` can evaluate to `True`, and analogously `F[i][j]` will tell us how many ways this subsection can evaluate to `False`.

The base case we can use to populate these matrices is when we are looking only at individual values, using the logic mentioned above:

```
if operand[i] == 'T':
    T[i][i] = 1; F[i][i] = 0
elif operand[i] == 'F'
    T[i][i] = 0; F[i][i] = 1
```

Next, we would like to consider larger and larger subproblems. For example, suppose our input is `F | T & T ^ F | T`, and we want to figure out the value of `T[0][3]`. This can be computed as the following sum:

```
(F) | (T & T ^ F) + (F | T) & (T ^ F) + (F | T & T) ^ F
```

In other words, for each possible split point in between `i` and `j`, which we can call k, we must accumulate the result of applying the operator at that split point to the two sides, each of which should already have been computed. To be more precise, the result will be a Cartesian product: that is, if there are two ways that the left side can evaluate to true, and three ways the right side can evaluate to true, then overall we should add six.

The only remaining logic is to figure out exactly how to apply these operators in order to update each matrix. We can look at each operator in turn to understand the correct behavior.

- For &, both sides must be true in order for the entire expression to be `True`.

Otherwise, it will be `False`.

- For `|`, both sides must be false in order for the entire expression to be `False`. Otherwise, it will be `True`.
- For `^`, the entire expression will be true if and only if one side is true and the other is false.

Finally, the value represented by `T[0][n - 1]` will tell us the number of ways the entire expression can evaluate to `True`.

We can put this together as follows, using a helper function called `split` to separate operators from operators in our input.

```
def split(expression):
    operands, operators = [], []

    for value in expression:
        if value in {'T', 'F'}:
            operands.append(value)
        else:
            operators.append(value)


    return operands, operators

def solve(expression):
    operands, operators = split(expression)

    n = len(operands)
    T = [[0 for _ in range(n)] for _ in range(n)]
    F = [[0 for _ in range(n)] for _ in range(n)]

    for i in range(n):
        if operands[i] == 'T':
            T[i][i] = 1; F[i][i] = 0
        else:
            T[i][i] = 0; F[i][i] = 1

    for gap in range(1, n):
```

```
        for i in range(n - gap):
            j = i + gap

            for k in range(i, j):
                all_options = (T[i][k] + F[i][k]) * (T[k+1][j] + F[k+1][j])

                if operators[k] == '&':
                    T[i][j] += T[i][k] * T[k+1][j]
                    F[i][j] += (all_options - T[i][j])

                elif operators[k] == '|':
                    F[i][j] += F[i][k] * F[k+1][j]
                    T[i][j] += (all_options - F[i][j])

                elif operators[k] == '^':
                    T[i][j] += F[i][k] * T[k+1][j] + T[i][k] * F[k+1][j]
                    F[i][j] += T[i][k] * T[k+1][j] + F[i][k] * F[k+1][j]

    return T[0][n - 1]
```

The space complexity will be $O(N^2)$, where N is the number of operands, in order to store the T and F matrices. Calculating each cell in both matrices requires us to calculate the result for each split point, which is $O(N)$, so the overall algorithm will take $O(N^3)$ time.