1896. Minimum Cost to Change the Final Value of Expression

Problem Explanation

Given a Boolean expression, the problem asks to calculate the minimum operations needed to flip or change the final output of the expression. Here, an operation is defined as replacing '&' with '|', or '|' with &', or flipping '0' to '1' or '1' to '0'. Parentheses are evaluated first and then calculation is done from left to right.

Example

```
python
Input: "(0&0)&(0&0&0)"
Output: 3
```

How did we get 3? Here, three operations are needed:

- Convert the first '0' to '1'. The expression becomes "(1&0)&(0&0&0)".
- Convert the second '0' after '&' to '1'. The expression becomes "(1&1)&(0&0&0)".
- Convert the third '0' after '&' to '1'. The expression becomes "(1&1)&(1&0&0)".

Thus, changing those three zeros to ones will result in the final value being 1.

Solution Approach

To solve this problem, we are using a stack data structure to maintain the changing states of the expression. We make use of three stacks, one for storing the result values, second for storing the minimal operations needed, and the third for storing operators.

As we parse through the expression, we push characters onto the stack. Each time we encounter a closing parenthesis or a digit, we pop from the stack and calculate corresponding costs.

ASCII illustration of the stack after three replacements:

```
stack after 1 operation -> [ '1', '1&0']
stack after 2 operations -> [ '1&1', '1&0']
stack after 3 operations -> [ '1&1&1', '0' ]
```

Python Solution

```
python
from typing import List

class Solution:
    def minOperationsToFlip(self, expression: str) -> int:
        stack = []
        for char in expression:
            if char in ('0', '1', '|', '&'):
                stack.append((char, 1 if char in '01' else 0))
            elif char == ')':
                stack.append(char)
            else:
                e2 = stack.pop()
                stack.pop()
                e1 = stack.pop()
                if e2[0] == '0' and e1[0] == '|':
                    stack.append(('0', 1))
                else:
                    stack.append((max(e1, e2, key=lambda x: (x[0] == '1', -x[1]))))
        while len(stack) > 1:
            e2 = stack.pop()
            op = stack.pop()
            e1 = stack.pop()
            if e2[0] == e1[0]:
                stack.append((e2[0], 1 + min(e1[1], e2[1])))
            else:
                stack.append((e2[0] if op[0] == '|' else e1[0], max(e1[1], e2[1])))
        return stack[0][1]
```

Explanation:

The Python solution uses stack to solve the problem. Looping through the expression, when '0', '1', '|', '&' are encountered, they are pushed onto the stack. When closing parenthesis is encountered it pops the last two expressions and decides the new expression and corresponding cost. The decision is made based on the bitwise rule. It then pushes the resulting expression back to the stack. We need to make sure that we handle case of braces and nested expressions separately. Hence, in the end of process, if the stack has more than one element, we pop out last two expressions and an operator, decide the cumulative expression and push it back to the stack. When there's only a single element left in the stack, we return its cost.

I encourage you to go through this problem with a debugger to understand it better. The stack is a great tool to "visualize" the state at each step.

Prime your brain and happy coding!# JavaScript Solution

```
javascript
class Solution {
    minOperationsToFlip = function(expression) {
        let chars = expression.split('');
        let stack = [];
        let map = {'0': '1', '1': '0', '&': '|', '|': '&'};

        for (let char of chars){
            if (char === ')'){
                let newStack = [];

                while (stack[stack.length - 1] !== '('){
                    newStack.push(stack.pop());
                }

                stack.pop();
                stack.push(this.getOperationsNeeded(newStack.reverse()));
            } else{
                stack.push(char);
            }
        }

        return this.getOperationsNeeded(stack);
    };

    getOperationsNeeded = function(chars){
        let map = {'0': '1', '1': '0', '&': '|', '|': '&'};
        let cost = new Map();
        cost.set('0', [0, 1]);
        cost.set('1', [1, 0]);

        let operations = [chars[0]];
        for (let i = 2; i < chars.length; i += 2){
            [operations[0], operations[1]] = this.getUpdatedCosts(chars[i - 1], operations, cost.get(chars[i]));
        }

        return map[operations[0]] + String(Math.max(operations[1], 1));
    }

    getUpdatedCosts = function(op, ops1, ops2){
        let map = {'0': '1', '1': '0', '&': '|', '|': '&'};

        if (ops1[0] === ops2[0]){
            return [ops1[0], Math.min(ops1[1], ops2[1]) + 1];
        }

        if (op === '|'){
            return [map[ops1[0]], Math.max(ops1[1], ops2[1])];
        } else{
            return [ops1[0], Math.max(ops1[1], ops2[1] + 1)];
        }
    }
}
```

Explanation:

The JavaScript solution uses a similar approach to the Python solution. It splits the input string into an array of characters and initializes a stack to hold the characters. A 'map' is created to make character replacements when needed. It then iterates through each character in the array and if a closing parenthesis is encountered, it pops all the characters from the stack until an opening parenthesis is found and pushed onto a new stack. It then replaces the earlier pushed characters with the minimum number of operations required to change the final result of the popped characters and pushes it back into the original stack. The getOperationsNeeded method is used to calculate the minimum number of operations needed to change the value of characters through the getUpdatedCosts method depending on the operators and digits encountered. This continues until all characters are traversed and finally returns the number of operations needed, which should be only the last element left in the stack.

Java Solution

```
java
import java.util.Stack;

class Solution {
    public int minOperationsToFlip(String expression) {
        Stack<Integer> stack = new Stack<>();
        int n = expression.length();
        for(int i = 0; i < n; ++i){
            if(expression.charAt(i) == '0' || expression.charAt(i) == '1'){
                stack.push(expression.charAt(i) - '0');
            }
            else if(expression.charAt(i) == '&'){
                stack.push(-2);
            }
            else if(expression.charAt(i) == '|'){
                stack.push(-1);
            }
            else if(expression.charAt(i) == ')'){
                int b = stack.pop();
                while(stack.peek() < 0){
                    int op = stack.pop();
                    int a = stack.pop();
                    if(op == -1){
                        int r = a | b;
                        stack.push(r);
                        b = r == 0 ? 2 : 1;
                    }
                    else{
                        int r = a & b;
                        stack.push(r);
                        b = r == 1 ? 2 : 1;
                    }
                }
                stack.pop();
                stack.push(b);
            }
        }
        int b = stack.pop();
        int ans = 0;
        while(!stack.isEmpty()){
            int op = stack.pop();
            int a = stack.pop();
            if(op == -1){
                ans = Math.max(Math.min(a, b), ans);
            }
            else{
                ans = Math.max(Math.min(a, 2 - b), ans);
            }
            b = Math.min(a, b);
        }
        return ans;
    }
}
```

Explanation:

The Java solution follows a similar approach as the Python and JavaScript solutions. It uses a Stack data structure to store characters of the Boolean expression. This solution process '0', '1', '(', '|' and '&' in the same way as the previous solutions.

This solution uses an while loop to process closing parenthesis ')'. In the while loop, it continues to pop elements from stack until an open parenthesis '(' is encountered. According to the 'op' value, it calculates the result and then pushes it back to the stack.