Boolean Parenthesization

DSA👻	
	53
topic 🦫	
🌰 Desc	Memorize this.

Why are we checking for false? We know from the question we want true, right?

You're absolutely right that the **main question** is:

> ❓ "How many ways to parenthesize the whole expression so that it evaluates to True?"

So then...

❓ Why are we checking for `False` values in recursion?

Because even though the **end goal** is to evaluate the entire expression to `True`, the **subproblems** (smaller expressions) might need to evaluate to either `True` or `False` — depending on what the operator in the middle is.

Let me break it down intuitively.

🔄 Example to Understand Why `False` is Needed

Take expression:

```
T ^ F
```

There is **only one operator**, and we want the result to be `True`.

Now `^` (XOR) is True **only when one operand is True and the other is False.**

So we must consider both:

- `T` (left) is `True`, `F` (right) is `False`
- AND vice versa (but not in this example since T is fixed)

If you didn't **calculate the number of ways to get** `False`, how would you evaluate the right side?

🤝 Recursive Subcalls Depend on Both Sides

Imagine this case:

```
T | F & T
```

You divide it at `|`:

- Left: `T`
- Right: `F & T`

Now, to evaluate the **right side (`F & T`)**, you must:

- Know how many ways it can become `True`
- And how many ways it can become `False`

Because:

- `T | anythingTrue` = `True`
- `T | anythingFalse` = `True`

But if **both** sides are False, `T | F = T` (still true)

So we **need both counts** (true and false) from both sides to compute all valid combinations.

🔁 Summary of Why We Need Both `True` and `False` Counts

- You can’t calculate how an operator affects subexpressions without **both** `True` and `False` outcomes
- Each operator (`&`, `|`, `^`) behaves differently based on left/right sides being True/False
- So, even if you want the final result to be `True`, your **recursive engine must support both outcomes**

🧠 Interview Analogy

Think of `isTrue` as a **contract** you pass to the recursive function:

> "Hey subproblem, I need you to give me the number of ways this part becomes True (or False), so I can combine it with others to fulfill my part of the expression."

🧩 Why do we need the `isTrue` boolean?

You're asking:

> "Since in the end we return the total ways that evaluate to True, why are we tracking isTrue at every step? Why can't we just always compute True cases?"

The short answer is:

> Because each recursive subproblem must be flexible enough to compute either True or False depending on what the parent problem needs.

🎯 Let’s take a real analogy first

Imagine solving a jigsaw puzzle.

- You know what the final picture looks like (you want `True`)

- But to solve smaller pieces (left/right subexpressions), sometimes you **need a piece to be shaped one way**, other times a **different shape**

> So isTrue is like saying:
>
> "Hey left half, can you form this exact shape (True or False)? Because that's what I need to make my part work."

🔄 Now from a recursion standpoint

Let's say you're solving:

```
countWays(s, i, j, True)
```

So you want to compute **ways to make** `s[i..j]` **become** `True`

Now suppose you split at operator `k`, say `|`.

You do:

```
leftTrue = countWays(s, i, k - 1, True)
leftFalse = countWays(s, i, k - 1, False)
rightTrue = countWays(s, k + 1, j, True)
rightFalse = countWays(s, k + 1, j, False)
```

Why so many calls?

Because the result of `a | b` is:

- `True` if **either** side is `True`
- `False` only if **both** are `False`

So, **in order to compute** `s[i..j] == True`, you need both:

- Left side being `True` AND right side being `anything`

- Left being `False`, right being `True`

That's why you can't just **always call for** `True`.

🧠 `isTrue` guides the recursion's goal

Without the `isTrue` flag, you wouldn't know:

- whether you're solving for a `True` result (combine using truth rules like `T & T`, `T | F`)
- or for a `False` result (combine with complementary rules like `F & T`, `F ^ F`, etc.)

Each subproblem must **adapt to what the parent is asking for** — and that's what `isTrue` tracks.

🛠️ Concrete Example

Let's break `T ^ F & T`

We're solving: `countWays(s, 0, 4, True)`

First split at `^` (index 1):

- Left: `T` (index 0) → must check how many ways it can be `True` and `False`
- Right: `F & T` (index 2 to 4) → must compute both `True` and `False` counts, because we don't know what is required unless we know the operator.

> If you didn't pass isTrue, then you couldn't tell the right side whether you need it to be True or False to satisfy your condition!

🔁 Summary

🔍 Question	✅ Why we need `isTrue`
Why not always calculate True?	Because sometimes the sub-expression needs to evaluate to **False** to make the parent expression `True` or `False` depending on the operator

🔍 Question	✅ Why we need `isTrue`
Why pass as an argument?	Because **each recursive subproblem** must compute **according to what's required by the operator above**
Is it just extra logic?	No. It is essential for correctly applying operator truth tables (`^`, `

Okay, so it's like the parent equation is asking the child, In how many ways can you be true, and in how many ways can you be false?, And based on that, the parent will calculate the number of ways it can be true, since even a false child can contribute to a true parent. Is my understanding correct?

🎯 **YES! That's exactly the right understanding, Shubham!**

You've nailed the core intuition behind this problem.

Let's reinforce it once more:

🔄 Your Summary (Restated Clearly):

> "The parent expression is asking each child (subexpression):
>
> 'In how many ways can you become `True`? In how many ways can you become `False`?'
>
> **Then, based on the operator**, it combines those answers to figure out how many ways **it** can become `True`."

✅ Perfect.

Even if the **parent wants to be `True`**, it might require:

- A `True` child and a `False` child (like in XOR `^`)
- Or even two `False` children (like in `False ^ False = False` for the opposite case)

So unless **every subproblem supports both outcomes (True and False)**, you can't properly calculate the overall result.

🧠 Visual Analogy

It's like each node in a tree asking its child:

> "I want to become True, can you help me? How many ways can you become True and False?"
>
> And based on the operator (`&`, `|`, `^`), the node decides how many of those combinations help it achieve its goal.

🧪 Small Example to Wrap Up

Expression: `T ^ F`

Goal: Evaluate to `True`

Break at `^`:

- Left (`T`) → ways to be True: 1, False: 0
- Right (`F`) → ways to be True: 0, False: 1

For `^`, True is achieved when exactly one side is True:

```
T ^ F = T
```

So total ways: `leftTrue * rightFalse + leftFalse * rightTrue = 1*1 + 0*0 = 1`

✅ And see how we **needed both True and False counts** to make that work.

🔍 Why are we doing multiplication like this?

```
ways += leftTrue * rightTrue
```

And **not just**:

```
ways += leftTrue + rightTrue
```

🧠 Because we're counting combinations of ways from left and right

This is **not** a simple sum of how many ways the left and right are `True` independently.

> We’re counting how many total ways the left and right expressions can be combined such that they satisfy the parent operator.

🧩 Let’s do a Simple Example

Let’s say:

- The left expression can become `True` in **2** ways
- The right expression can become `True` in **3** ways
- The parent operator is `&`
- And we want to know: **how many total ways can this whole thing become `True`**

Now:

For `A & B` to be `True`, **both** `A` and `B` must be `True`

So for each of the **2** `True` **ways** on the left, it can be paired with **3** `True` **ways** on the right.

✅ Total combinations = `2 * 3 = 6`

So:

```
python
CopyEdit
ways += leftTrue * rightTrue
```

🧱 Why Not Addition?

If you did:

```
ways += leftTrue + rightTrue  # 2 + 3 = 5 ❌
```

You're just adding counts, but you're **not accounting for combinations**.

It would be like saying:

> "Oh I have 2 ways on the left, and 3 on the right, so maybe 5 total?"
>
> ❌ That's logically incorrect because you're **pairing one left with many rights** — it's a **Cartesian Product** scenario.

🥪 Real-life Analogy

Imagine you're choosing:

- 2 types of **bread**
- 3 types of **filling**

And want to know: **How many different sandwiches you can make?**

Each bread can go with every filling → `2 * 3 = 6` sandwiches

✅ Not 2 + 3 = 5

Same logic applies in Boolean Parenthesization!

✅ Final Summary

Concept	Explanation
Why multiply?	We're counting **pairwise combinations** of left and right outcomes that satisfy the parent logic
When do we multiply?	Anytime we combine results from left × right — always combinations
When would we add?	When summing different partitions — like trying all split points in a loop

---

Code

🔍 Problem Statement Recap

Given a boolean expression made up of:

- `'T'` and `'F'` as operands
- `&`, `|`, `^` as operators

We need to count **how many ways** we can parenthesize the expression so that the **whole expression evaluates to** `True`.

🧠 Key Intuition Before Recursion

We will solve this by **dividing** the expression at every operator and **recursively** solving the left and right parts.

At each recursive step:

- You decide a partition point at an operator (say `&`)
- You recursively compute:
  - Number of ways left part becomes True / False
  - Number of ways right part becomes True / False

- Then you combine results **based on the operator** logic.

✅ Recursive Function Signature

```
def countWays(s, i, j, isTrue):
```

- `s`: the expression string
- `i`, `j`: current substring indices (inclusive)
- `isTrue`: boolean — do we want this substring to evaluate to True or False?

🔁 Recursive Base Case

```
if i > j:
    return 0

if i == j:
    if isTrue:
        return 1 if s[i] == 'T' else 0
    else:
        return 1 if s[i] == 'F' else 0
```

Explanation:

- When `i == j`, it means the substring is just one character — `'T'` or `'F'`
- If we want it to be `True`, we return 1 only if it's `'T'`

🔧 Recursive Case: Break at Every Operator

```
ways = 0
```

```
for k in range(i + 1, j, 2):  # k points to operator
    leftTrue = countWays(s, i, k - 1, True)
    leftFalse = countWays(s, i, k - 1, False)
    rightTrue = countWays(s, k + 1, j, True)
    rightFalse = countWays(s, k + 1, j, False)

    op = s[k]

    if op == '&':
        if isTrue:
            ways += leftTrue * rightTrue
        else:
            ways += leftFalse * rightTrue + leftTrue * rightFalse + leftFalse * rightF
alse
    elif op == '|':
        if isTrue:
            ways += leftTrue * rightTrue + leftTrue * rightFalse + leftFalse * rightTr
ue
        else:
            ways += leftFalse * rightFalse
    elif op == '^':
        if isTrue:
            ways += leftTrue * rightFalse + leftFalse * rightTrue
        else:
            ways += leftTrue * rightTrue + leftFalse * rightFalse
```

🔑 Operator Truth Table (Important to Memorize)

Operator	T op T	T op F	F op T	F op F
&	T	F	F	F
`	`	T	T	T
^	F	T	T	F

We use this table to determine how many combinations give us the desired result.

🧪 Example Walkthrough

Expression: `'T|F&T'`

Goal: Count number of ways to parenthesize so the result is True

- First divide at index 1 (`|`)
  - Left: `'T'`
  - Right: `'F&T'` (recursively solved)

Then divide `'F&T'` at index 3 (`&`) — and continue solving recursively.

🧵 Full Recursive Code (No Memoization Yet)

```
def countWays(s, i, j, isTrue):
    if i > j:
        return 0

    if i == j:
        if isTrue:
            return 1 if s[i] == 'T' else 0
        else:
            return 1 if s[i] == 'F' else 0

    ways = 0
    for k in range(i + 1, j, 2):  # k is operator index
        leftTrue = countWays(s, i, k - 1, True)
        leftFalse = countWays(s, i, k - 1, False)
        rightTrue = countWays(s, k + 1, j, True)
        rightFalse = countWays(s, k + 1, j, False)

        op = s[k]

        if op == '&':
            if isTrue:
                ways += leftTrue * rightTrue
```

```
            else:
                ways += leftFalse * rightTrue + leftTrue * rightFalse + leftFalse * righ
tFalse
        elif op == '|':
            if isTrue:
                ways += leftTrue * rightTrue + leftTrue * rightFalse + leftFalse * right
True
            else:
                ways += leftFalse * rightFalse
        elif op == '^':
            if isTrue:
                ways += leftTrue * rightFalse + leftFalse * rightTrue
            else:
                ways += leftTrue * rightTrue + leftFalse * rightFalse

    return ways
```

⏱️ Time Complexity (Recursion only)

At each operator, we branch 4 times (True/False × True/False), and there are `O(n)` operator positions.

Time: O(4ⁿ) — exponential.

🚧 Tabulation Plan: Step-by-Step

Tabulation is harder here due to the extra dimension (`isTrue`) and the need to split on operators.

So let's first clarify how to **build a DP table** for this problem.

🧠 1. DP Table Structure

We'll define:

```
dp[i][j][isTrue]
```

- `i` to `j` → the substring of the expression
- `isTrue` → whether we want it to evaluate to `True (1)` or `False (0)`
- So `dp[i][j][1]` = number of ways `s[i..j]` evaluates to True
- And `dp[i][j][0]` = number of ways `s[i..j]` evaluates to False

📦 2. Initialize DP Table

```
n = len(s)
dp = [[[0 for _ in range(2)] for _ in range(n)] for _ in range(n)]
```

⚙️ 3. Base Case Initialization

If `i == j`:

- If `s[i] == 'T'`:

  `dp[i][i][1] = 1` and `dp[i][i][0] = 0`

- If `s[i] == 'F'`:

  `dp[i][i][0] = 1` and `dp[i][i][1] = 0`

This handles single characters.

🔁 4. Fill DP Table (Bottom-Up)

We build up from smaller substrings to larger ones:

```
#User function Template for python3
class Solution:
    def countWays(self, s):
```

```
        # code here
        n = len(s)

        # isTrue = True or False

        dp = [[[0 for _ in range(2)] for _ in range(n)]for _ in range(n)]

        # base case
        for i in range(0, n, 2):  # Only operands
            if s[i] == 'T':
                dp[i][i][1] = 1 # isTrue
            else:
                dp[i][i][0] = 1

        # i = 0 → n
        # j = n - 1 → 0

        for i in reversed(range(0, n, 2)):
            for j in range(i+2, n, 2): # j must be greater than i
                for isTrue in range(2):
                    ways = 0
                    for k in range(i+1, j, 2):
                        leftTrue = dp[i][k - 1][1]# in how many ways can left child be True
                        leftFalse = dp[i][k - 1][0] # in how many ways can left child be Fals
                        rightTrue = dp[k+1][j][1] # in how many ways can right child be Tru
                        rightFalse = dp[k+1][j][0] # in how many ways can right child be Fa

                        op = s[k]

                        if op == '&':
                            if isTrue:
                                # only way for and to be true is if left and right child are true
                                ways += leftTrue * rightTrue
                            else:
                                # False = 0 & 0, 0 & 1,  1 & 0
                                ways += (leftFalse * rightFalse) + (leftFalse * rightTrue) + (left
```

```
                elif op == '|':
                    if isTrue:
                        # True: 1 or 1, 1 or 0 , 0 or 1
                        ways += (leftTrue * rightTrue) + (leftTrue * rightFalse) + (leftF
                    else:
                        # False : 0 or 0
                        ways += leftFalse * rightFalse
                elif op == '^':
                    if isTrue:
                        # True = 1 ^ 0 , 0 ^ 1
                        ways += (leftTrue * rightFalse) + (leftFalse * rightTrue)
                    else:
                        # False = 0 ^ 0, 1 ^ 1
                        ways += (leftTrue * rightTrue) + (leftFalse * rightFalse)

            dp[i][j][isTrue] = ways

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

🧾 5. Return the Result

Finally,

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

🧠 Summary

Step	Description
`dp[i][j][1]`	# of ways `s[i..j]` becomes True
`dp[i][j][0]`	# of ways `s[i..j]` becomes False
Base case	Initialize for single characters
Transition	Use truth tables based on operator `&` , `
Final result	`dp[0][n - 1][1]`