Previous Next

150. Evaluate Reverse Polish Notation

Feb. 15, 2020 | 9.5K views

Average Rating: 4.86 (29 votes)

Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are `+`, `-`, `*`, `/`. Each operand may be an integer or another expression.

Note:

- Division between two integers should truncate toward zero.
- The given RPN expression is always valid. That means the expression would always evaluate to a result and there won't be any divide by zero operation.

Example 1:

```
Input: ["2", "1", "+", "3", "*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: ["4", "13", "5", "/", "+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

Example 3:

```
Input: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
Output: 22
Explanation:
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

Solution

If you've attempted this question and can't figure out why you're getting wrong answers, here are a couple of things to check *before* reading the article:

1. Reverse Polish Notation **is not a "reverse" form of Polish Notation**. It is a bit different.
2. If you're using Java, note that **the input type is an array of strings**, *not* an array of chars. This means that you should be comparing them with `.equals(...)`, *not* `==`. If your code is working on your computer but not on Leetcode, this is probably why. *It is a bug in your code, not in the Leetcode platform.*
3. **Some programming languages** (e.g. Python, but not C++ and Java) **do not truncate towards 0 with division**, so you'll need to figure out how to make them do so (we'll discuss ways in the article). For example, if we put `-121 // 7` into Python, we get `-18`, but we actually wanted `-17`. If unsure about your programming language, either check the documentation or simply write a program that does `-121 / 7` (as integer division) and see which result you get.
4. **Did you put numbers around the correct way?** e.g. the test case `["12", "7", "-"]` means you should calculate `12 - 7 = 5`, and ***not*** `7 - 12 = -5`. In most implementations, getting them the correct way around is not immediately obvious. If you aren't *certain* you have it right, try this test case (`["12", "7", "-"]`) and check whether you get `5` or `-5`.
5. At the time of writing this solution article, the Wikipedia article has a number of errors and sections that are confusing (in particular, parsing the list in reverse). Try to understand how Reverse Polish Notation works and then design an algorithm yourself rather than following the provided pseudocode too closely. The Introduction section of this solution article also explains how Reverse Polish Notation works.

Introduction

We'll start with looking at what it means for integer division to truncate towards zero, and then at what Reverse Polish Notation is. Feel free to skip these sections if you already know this stuff.

Division between two integers should truncate towards zero

Early on in your programming career, you probably learned about integer division. When dividing 2 positive numbers, we always *truncate down* to the nearest integer. The non-integer values are in parenthesis afterwards for reference.

```
6 / 2 = 3 (3.0)
11 / 5 = 2 (2.2)
9 / 5 = 1 (1.8)
```

Most programming languages do integer division by default (as opposed to float division, where decimal places are kept), and this is how they handle positive integers. Both of the following definitions could be (and are) used to describe the truncation.

1. The result is truncated to a *less than or equal number*. i.e. `1` is less than `1.8`.
2. The truncation is towards zero, i.e. `1` is closer to zero than `1.8` is.

For negative numbers however, it is *impossible* to satisfy both of these, so one or the other has to be picked. For example, consider the following:

```
-9 / 5 = ? (-1.8)
```

1. If we wanted the truncated result to be *smaller*, we'd have to go to `-2`, as `-2 < -1`.
2. If we wanted the truncated result to be *nearer to zero*, we'd have to go to `-1` as `-1` is nearer to zero than `-2` is.

Some programming languages go with the first definition, and others go with the second. For this problem, you are expected to go with the second definition, regardless of what your chosen programming language uses.

Python, for example, goes with the first definition. This means that we need to find a way of doing the division. Luckily, the `int(...)` function *does* truncate towards zero, and therefore we can use the `int(a / b)` trick. Note that this is *not* the same as `int(a // b)`. We haven't checked what all the programming languages available on Leetcode do, so if your chosen programming language is not truncating division towards zero, have a look in the math libraries for your chosen programming language or do a web search.

What is Infix Notation?

Analysing Infix Notation provides some great context for understanding Reverse Polish Notation.

Most people know how to read expressions written using Infix Notation. Evaluating the following expression is something you will have learned to do in elementary school.

```
3 + 1 + 9 - 5 = 8
```

This isn't too difficult. However, many of you will also have seen viral posts circulating social media websites, such as Facebook, that challenge you to evaluate an expression like:

```
5 * 4 + 9 - 2 / 3 + 1 = ?
```

When you check the comments, among many other strange answers, you'll probably see people arguing about whether the answer is `10`, `28.33`, or `29.33`.

The reason for the disagreement is because different people have different understandings about how such an expression should be evaluated.

Those who say the answer is `10` evaluated it strictly from left to right, with the following steps:

```
5 * 4 = 20
20 + 9 = 29
29 - 2 = 27
27 / 3 = 9
9 + 1 = 10
```

Those who say the answer is `28.33` follow a rule where we evaluate operations in the following order; division, multiplication, addition, and subtraction. This method comes from a common misunderstanding of the widely used mnemonics: PEMDAS/BODMAS/BEDMAS. The steps with this method are as follows:

```
= 5 * 4 + 9 - 0.66 + 1 (Do division first.)
= 20 + 9 - 0.66 + 1 (Do multiplication second.)
= 29 - 1.66 (Do the additions third.)
= 28.33 (Do the subtraction fourth.)
```

Those who say the answer is `29.33` use the rules most programming languages use, and that is also the correct interpretation of the mnemonics (PEMDAS/BODMAS/BEDMAS). That is to do division *and* multiplication first, in order from left to right, and then addition *and* subtraction, in order from left to right. Their steps are as follows:

```
= 20 + 9 - 2 / 3 + 1 (Do the multiplication.)
= 20 + 9 - 0.66 + 1 (Do the division.)
= 29 - 0.66 + 1 (Do the first addition.)
= 28.33 + 1 (Do the subtraction, as it's next.)
= 29.33 (Do the last addition.)
```

Most mathematicians would agree that the correct answer is `29.33`. Yet, this is probably not the answer you'd get if you asked a random sample of people in the general public (just look at the Facebook posts!). In school for example, I was taught the method that gives `28.33`. It wasn't until I learned programming in university that I learned the correct way!

When we want to do the operations in a different order, we use parenthesis (brackets) around the parts to do first. The parts in parenthesis are always done before the parts outside.

The big *disadvantage* of **Infix Notation** is hopefully clear now. The rules for evaluating it are surprisingly complex, cause a lot of confusion, and in fact most people don't understand them properly. Additionally, the need to use parentheses correctly adds another layer of complexity.

As we move towards understanding what **Reverse Polish Notation** is, keep in mind that while it seems a bit strange and un-intuitive (at first!), that **Infix Notation** is actually more confusing. The *only* reason Infix Notation seems intuitive is because you've probably been using it all your life and so it is now second nature to you. People who use Reverse Polish Notation on a daily basis find it very intuitive! Some hand-held calculators still use it!

What is Reverse Polish Notation?

Just like Infix Notation, or in fact any other notation, **Reverse Polish Notation** has rules for how to evaluate it. You'll need to know these rules before you can write an algorithm. The rules could either be prior knowledge or supplied by an interviewer.

> While there are operators remaining in the list, find the left-most operator. Apply it to the 2 numbers immediately before it, and replace all 3 tokens (the operator and 2 numbers) with the result.

For example in the most simplest case of `3 4 +` when we reach `+` we can replace `3 4 +` with it's result `7`.

As long as the input was valid, this rule will always work and leave a *single number* that should be returned. The leftmost operator that hasn't yet been removed will *always* have 2 numbers immediately before it.

Here is an animation showing a more complicated example.

Hopefully the advantage is obvious now. Reverse Polish Notation doesn't require brackets, and the rules for evaluating it are far simpler. In-fact, our equivalent question for Infix Notation is much more difficult than this one!

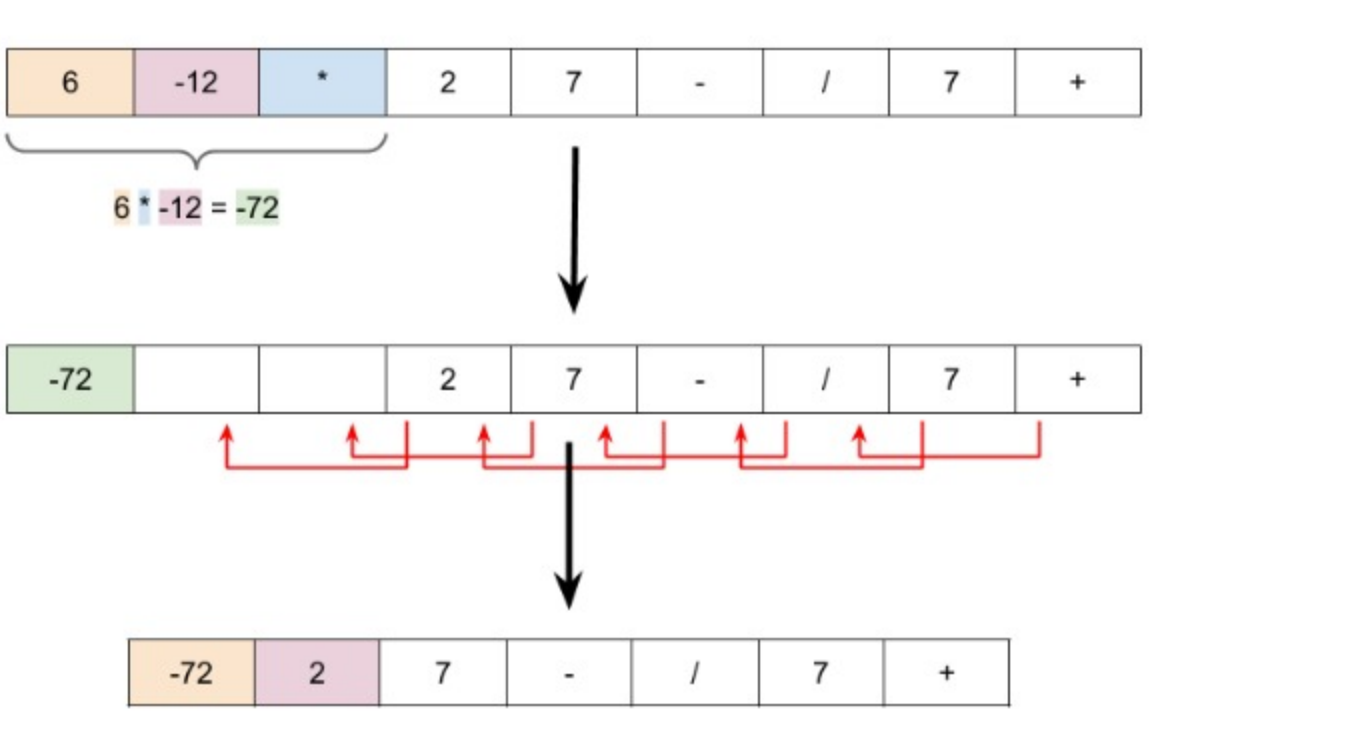
Approach 1: Reducing the List In-place

Intuition

This approach literally follows the animation above. A pointer is used to step through the list, and each time an operator (`+-*/) is found, that operator is then applied to the 2 values before it (which are always numbers, as long as the original input was valid). The 3 values are then replaced with the result. This process is repeated until the list is of length 1, containing a single number that is the answer to be returned.

Algorithm

The code is simpler for Python than Java. For Java (and other languages where the input type is a fixed size array), we have to define our own method to delete values from an array. This is done by shuffling the other elements down into the gap. Of course, you could start by copying the input into an `ArrayList` (so that you could then use its delete method), but then the algorithm would require $O(n)$ space instead of $O(1)$ space.



We have to be a little careful about the types in the Python code, as in the middle of processing the list, some numbers will be represent as strings, and others as ints. Additionally, we also need to be aware that *python division does not truncate towards zero*. We can instead use `int(a / b)` to achieve the desired result. Note that this is *not* the same as `int(a // b)`.

In the Java code, we have to convert ints back to Strings, because Java doesn't support mixed lists of Strings and ints.

Finally, If you know how to use `lambda` functions in your chosen programming language, an ideal solution would use them to elegantly handle the 4 operations (`+-*/`). The first set of solutions here use `lambda` functions. If you aren't familiar with `lambda` functions though (or your chosen programming language doesn't support them), that's fine. You'll get a chance to learn them when you're ready! Scroll down a little more, as we have also provided solutions that don't use them.

Java | Python | JavaScript — Copy

```python
        }

        current_position = 0

        while len(tokens) > 1:

            # Move the current position pointer to the next operator.
            while tokens[current_position] not in "+-*/":
                current_position += 1

            # Extract the operator and numbers from the list.
            operator = tokens[current_position]
            number_1 = int(tokens[current_position - 2])
            number_2 = int(tokens[current_position - 1])

            # Calculate the result to overwrite the operator with.
            operation = operations[operator]
            tokens[current_position] = operation(number_1, number_2)

            # Remove the numbers and move the pointer to the position
            # after the new number we just added.
            tokens.pop(current_position - 2)
            tokens.pop(current_position - 2)
            current_position -= 1

        return tokens[0]
```

Here are the solutions without the use of `lambda` functions.

Java | Python | JavaScript — Copy

```python
def evalRPN(self, tokens: List[str]) -> int:

    current_position = 0

    while len(tokens) > 1:

        # Move the current position pointer to the next operator.
        while tokens[current_position] not in "+-*/":
            current_position += 1

        # Extract the operator and numbers from the list.
        operator = tokens[current_position]
        number_1 = int(tokens[current_position - 2])
        number_2 = int(tokens[current_position - 1])

        # Calculate the result to overwrite the operator with.
        if operator == "+":
            tokens[current_position] = number_1 + number_2
        elif operator == "-":
            tokens[current_position] = number_1 - number_2
        elif operator == "*":
            tokens[current_position] = number_1 * number_2
        else:
            tokens[current_position] = int(number_1 / number_2)

        # Remove the numbers and move the pointer to the position
        # after the new number we just added.
```

Complexity Analysis

Let n be the length of the list.

- Time Complexity : $O(n^2)$.

  Firstly, it helps to calculate how many *operators* and how many *numbers* are in the initial list. Each step of the algorithm removes 1 operator, 2 numbers, and adds back 1 number. This is an overall loss of 1 number and 1 operator per step. At the end, we have 1 number left. Therefore, we can infer that at the start, there must always be **exactly 1 more number than there is operators**.

  The big inefficiency of this approach is more obvious in the Java code than the Python. Deleting an item from an ArrayList or Array is $O(n)$, because all the items after have to be shuffled down one place to fill in the gap. The number of these deletions we need to do is the same as the number of operators, which is proportional to n. Therefore, the cost of the deletions is $O(n^2)$.

  This is more obvious in the Java code, because we had to define the deletion method ourselves. However, the Python deletion method works the same way, it's just that you can't see it because it's hidden in a library function call. It's important to always be aware of the cost of library functions as they can sometimes look like they're $O(1)$ when they're not!

- Space Complexity : $O(1)$.

  The only extra space used is a constant number of single-value variables. Therefore, the overall algorithm requires $O(1)$ space.

Interestingly, this approach could be adapted to work with a **Double-Linked List**. It would require $O(n)$ space to create the list, and then take $O(n)$ time to process it using a similar algorithm to above. This works because the algorithm is traversing the list in a linear fashion and modifications only impact the tokens immediately to the left of the current token.

Approach 2: Evaluate with Stack

Intuition

The first approach worked, but $O(n^2)$ is too slow for large n. As hinted at above, a **Double-Linked List** could be an option. However, it requires a lot of set-up code, and in practice requires more space than the elegant **Stack** approach we're going to look at now.

We don't want to repeatedly delete items from the middle of a list, as this inevitably leads to $O(n^2)$ time performance. So recall that the above algorithm scanned through the list from left to right, and each time it reached an operator, it'd replace the operator and the 2 numbers immediately before it with the result of applying the operator to the 2 numbers.

The two key steps of the above algorithm were:

1. Visit each operator, in linear order. Finding these can be done with a linear search of the original list.
2. Get the 2 most recently seen numbers that haven't yet been replaced. These could be tracked using a **Stack**.

The algorithm would be as follows:

```
stack = new Stack()
for each token in tokens:
    if token is a number:
        stack.push(token)
    else (token is operator):
        number_2 = stack.pop()
        number_1 = stack.pop()
        result = apply_operator(token, number_1, number_2)
        stack.push(result)
return stack.pop()
```

Here is an animation showing the algorithm.

You might have noticed the following 2 lines of the pseudocode could look like they're around the wrong way.

```
number_2 = stack.pop()
number_1 = stack.pop()
```

They are correct though. Remember that for division and subtraction, the order of the numbers matters. i.e. $7 - 5 \neq 5 - 7$. On the `Stack`, we have the *second* on the top. So we need to reverse them before applying the operator.

Algorithm

Here is code that uses `lambda` functionality. Scroll down for code that doesn't.

Java | Python | JavaScript — Copy

```python
def evalRPN(self, tokens: List[str]) -> int:

    operations = {
        "+": lambda a, b: a + b,
        "-": lambda a, b: a - b,
        "/": lambda a, b: int(a / b),
        "*": lambda a, b: a * b
    }

    stack = []
    for token in tokens:
        if token in operations:
            number_2 = stack.pop()
            number_1 = stack.pop()
            operation = operations[token]
            stack.append(operation(number_1, number_2))
        else:
            stack.append(int(token))
    return stack.pop()
```

Here are the solutions without `lambda`.

Java | Python | JavaScript — Copy

```python
def evalRPN(self, tokens):

    stack = []

    for token in tokens:

        if token not in "+-/*":
            stack.append(int(token))
            continue

        number_2 = stack.pop()
        number_1 = stack.pop()

        result = 0
        if token == "+":
            result = number_1 + number_2
        elif token == "-":
            result = number_1 - number_2
        elif token == "*":
            result = number_1 * number_2
        else:
            result = int(number_1 / number_2)

        stack.append(result)

    return stack.pop()
```

Complexity Analysis

Let n be the length of the list.

- Time Complexity : $O(n)$.

  We do a linear search to put all numbers on the stack, and process all operators. Processing an operator requires removing 2 numbers off the stack and replacing them with a single number, which is an $O(1)$ operation. Therefore, the total cost is proportional to the length of the input array. Unlike before, we're no longer doing expensive deletes from the middle of an Array or List.

- Space Complexity : $O(n)$.

  In the worst case, the stack will have all the numbers on it at the same time. This is never more than half the length of the input array.

Rate this article:

Previous Next

Comments: 5 Sort By

Type comment here... (Markdown is supported)

Preview Post

ramdoss ★ 7 February 15, 2020 7:02 PM
Excellent Article
7 Share Reply

ihsanmpm ★ 7 February 16, 2020 5:03 PM
great, detailed solution.
5 Share Reply

ultraInstinct ★ 21 April 30, 2020 11:44 AM
Using Deque instead of Stack is recommended for Java solution
1 Share Reply
SHOW 1 REPLY

nadaralp ★ 5 2 hours ago
Amazing article !.
I loved the dictionary with the lambda trick in Python
0 Share Reply

kremebrulee ★ 52 May 10, 2020 4:47 AM
approach 1 destroys the input array. To be able to reconstruct the input, you HAVE to allocate additional space. Therefore, it's better to perform deletions on a new linked list.
0 Share Reply