Previous Next

301. Remove Invalid Parentheses

Oct. 3, 2018 | 148.6K views

Average Rating: 3.85 (92 votes)

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses `(` and `)`.

Example 1:

```
Input: "()())()"
Output: ["()()()", "(())()"]
```

Example 2:

```
Input: "(a)())()"
Output: ["(a)()()", "(a())()"]
```

Example 3:

```
Input: ")("
Output: [""]
```

Solution

Approach 1: Backtracking

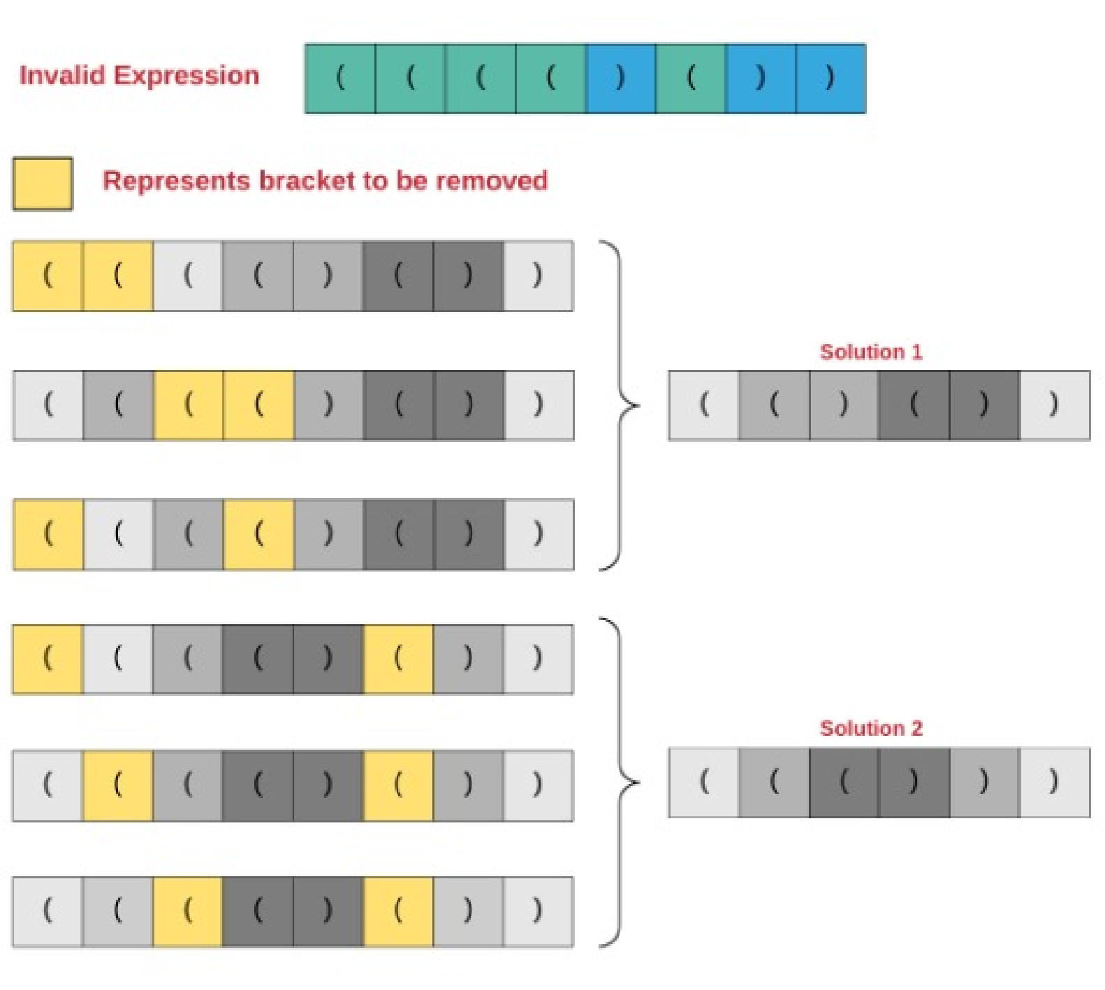
Intuition

For this question, we are given an expression consisting of parentheses and there can be some misplaced or extra brackets in the expression that cause it to be invalid. An expression consisting of parentheses is considered valid only when every closing bracket has a corresponding opening bracket and vice versa.

This means if we start looking at each of the bracket from left to right, as soon as we encounter a closing bracket, there should be an unmatched opening bracket available to match it. Otherwise the expression would become invalid. The expression can also become invalid if the number of opening parentheses i.e. `(` are more than the number of closing parentheses i.e. `)`.

Let us look at an invalid expression and all the possible valid expressions that can be formed from it by removing some of the brackets. There is no restriction on which parentheses we can remove. We simply have to make the expression valid.

> The only condition is that we should be removing the minimum number of brackets to make an invalid expression, valid. If this condition was not present, we could potentially remove most of the brackets and come down to say 2 brackets in the end which form `()` and that would be a valid expression.



An important thing to observe in the above diagram is that there are multiple ways of reaching the same solution i.e. say the optimal number of parentheses to be removed to make the original expression valid is K. We can remove multiple different sets of K brackets that will eventually give us the same final expression. But, each valid expression should be recorded only once. We have to take care of this in our solution. Note that there are other possible ways of reaching one of the two valid expressions shown above. We have simply shown 3 ways each for the two valid expressions.

Coming back to our problem, the question that now arises is, how to decide which of the parentheses to remove?

> Since we don't know which of the brackets can possibly be removed, we try out all the options!

For every bracket we have two choices:

- Either it can be considered a part of the final expression OR
- It can be ignored i.e. we can delete it from our final expression.

Such kind of problems where we have multiple options and we have no strategy or metric of deciding greedily which option to take, we try out all of the options and see which ones lead to an answer. These type of problems are perfect candidates for the programming paradigm, `Recursion`.

Algorithm

1. Initialize an array that will store all of our valid expressions finally.
2. Start with the leftmost bracket in the given sequence and proceed right in the recursion.
3. The state of recursion is defined by the index which we are currently processing in the original expression. Let this index be represented by the character `i`. Also, we have two different variables `left_count` and `right_count` that represent the number of left and right parentheses we have added to our expression till now. These are the parentheses that were considered.
4. If the current character i.e. `S[i]` (considering S is the expression string) is neither a closing or an opening parenthesis, then we simply add this character to our final solution string for the current recursion.
5. However, if the current character is either of the two brackets i.e. `S[i] == '(' or S[i] == ')'`, then we have two options. We can either discard this character by marking it an invalid character or we can consider this bracket to be a part of the final expression.
6. When all of the parentheses in the original expression have been processed, we simply check if the expression represented by `expr` i.e. the expression formed till now is valid one or not. The way we check if the final expression is valid or not is by looking at the values in `left_count` and `right_count`. For an expression to be valid `left_count == right_count`. If it is indeed valid, then it could be one of our possible solutions.
    - Even though we have a valid expression, we also need to keep track of the number of removals we did to get this expression. This is done by another variable passed in recursion called `rem_count`.
    - Once recursion finishes we check if the current value of `rem_count` is < the least number of steps we took to form a valid expression till now i.e. the global minima. If this is not the case, we don't record the new expression, else we record it.

One small optimization that we can do from an implementation perspective is introducing some sort of pruning in our algorithm. Right now we simply go till the very end i.e. process all of the parentheses and when we are done processing all of them, we check if the expression we have can be considered or not.

We have to wait till the very end to decide if the expression formed in recursion is a valid expression or not. Is there a way for us to cutoff from some of the recursion paths early on because they wouldn't lead to a solution? The answer to this is Yes! The optimization is based on the following idea.

For a left bracket encountered during recursion, if we decide to consider it, then it may or may not lead to an invalid final expression. It may lead to an invalid expression eventually if there are no matching closing bracket available afterwards. But, we don't know for sure if this will happen or not.

> However, for a closing bracket, if we decide to keep it as a part of our final expression (remember for every bracket we have two options, either to keep it or to remove it and recurse further) and there is no corresponding opening bracket to match it in the expression till now, then it will definitely lead to an invalid expression no matter what we do afterwards.

e.g.

```
( ( ) ) )
```

In this case the third closing bracket will make the expression invalid. No matter what comes afterwards, this will give us an invalid expression and if such a thing happens, we shouldn't recurse further and simply prune the recursion tree.

That is why, in addition to having the index in the original string/expression which we are currently processing and the expression string formed till now, we also keep track of the number of left and right parentheses. Whenever we keep a left parenthesis in the expression, we increment its counter. For a right parenthesis, we check if `right_count < left_count`. If this is the case then only we consider that right parenthesis and recurse further. Otherwise we don't as we know it will make the expression invalid. This simple optimization saves a lot of runtime.

Now, let us look at the implementation for this algorithm.

```
class Solution(object):

    def __init__(self):
        self.valid_expressions = None
        self.min_removed = None

    def reset(self):
        self.valid_expressions = set()
        self.min_removed = float("inf")

    """
        string: The original string we are recursing on.
        index: current index in the original string.
        left: number of left parentheses till now.
        right: number of right parentheses till now.
        ans: the resulting expression in this particular recursion.
        ignored: number of parentheses ignored in this particular recursion.
    """
    def remaining(self, string, index, left_count, right_count, expr, rem_count):
        # If we have reached the end of string.
        if index == len(string):

            # If the current expression is valid. The only scenario where it can be
            # invalid here is if left > right. The other way around we handled early on in the recursion.
            if left_count == right_count:

```

Complexity analysis

- Time Complexity : $O(2^N)$ since in the worst case we will have only left parentheses in the expression and for every bracket we will have two options i.e. whether to remove it or consider it. Considering that the expression has N parentheses, the time complexity will be $O(2^N)$.
- Space Complexity : $O(N)$ because we are resorting to a recursive solution and for a recursive solution there is always stack space used as internal function states are saved onto a stack during recursion. The maximum depth of recursion decides the stack space used. Since we process one character at a time and the base case for the recursion is when we have processed all of the characters of the expression string, the size of the stack would be $O(N)$. Note that we are not considering the space required to store the valid expressions. We only count the intermediate space here.

Approach 2: Limited Backtracking!

Although the previous solution does get accepted on the platform, it is a very inefficient solution because we try removing each and every possible parentheses from the expression and in the end we check two things:

1. if the expression is valid or not
2. if the total number of removed parentheses removed in the current recursion is less than the global minimum till now or not.

We cannot determine which of the parentheses are misplaced because, as the problem statement puts across, we can remove multiple combinations of parentheses and end up with a valid expression. This means there can be multiple valid expressions from a single invalid expression and we have to find all of them.

> The one thing all these valid expressions have in common is that they will all be of the same length i.e. as compared to the original expression, all of these expressions will have the same number of characters removed.

What if we could determine this count?

What if in addition to determining this count of characters to be removed, we could also determine the number of left parentheses and number of right parentheses to be removed from the original expression to get **any** valid expression?

This would cut down the computations immensely and the runtime would plummet as a result. The reason for this is, if we knew how many left and right parentheses are to be removed from the original expression to get a valid expression, we would cut down on so many unwanted recursive calls.

Imagine the original expression to be 1000 characters with only 3 misplaced `(` parentheses and 2 misplaced `)` parentheses. In our previous solution we would end up trying to remove each one of left and right parentheses and try to reach a valid expression in the end whereas we should only be trying out removing 3 `(` brackets and 2 `)` brackets.

> This is the exact number of `(` and `)` that have to be removed to get a valid expression. No more, no less.

Let us look at how we can find out the number of misplaced left and right parentheses in a given expression first and then we will slightly modify our original algorithm to incorporate these counts as well.

1. We process the expression one bracket at a time starting from the left.
2. Suppose we encounter an opening bracket i.e. `(`, it may or may not lead to an invalid expression because there can be a matching ending bracket somewhere in the remaining part of the expression. Here, we simply increment the counter keeping track of left parentheses till now. `left += 1`
3. If we encounter a closing bracket, this has two meanings:
    - Either there was no matching opening bracket for this closing bracket and in that case we have an invalid expression. This is the case when `left == 0` i.e. when there are no unmatched left brackets available. In such a case we increment another counter say `right += 1` to represent misplaced right parentheses.
    - Or, we had some unmatched opening bracket available to match this closing bracket. This is the case when `left > 0`. In this case we simply decrement the left counter we had i.e. `left -= 1`
4. Continue processing the string until all parentheses have been processed.
5. In the end the values of `left` and `right` would tell us the number of unmatched `(` and `)` parentheses respectively.

Now that we have these two values available that tell us the total number of left i.e. `(` and right i.e. `)` parentheses that have to be removed to make the invalid expression valid, we will modify our original algorithm discussed in the previous session to avoid unwanted recursions.

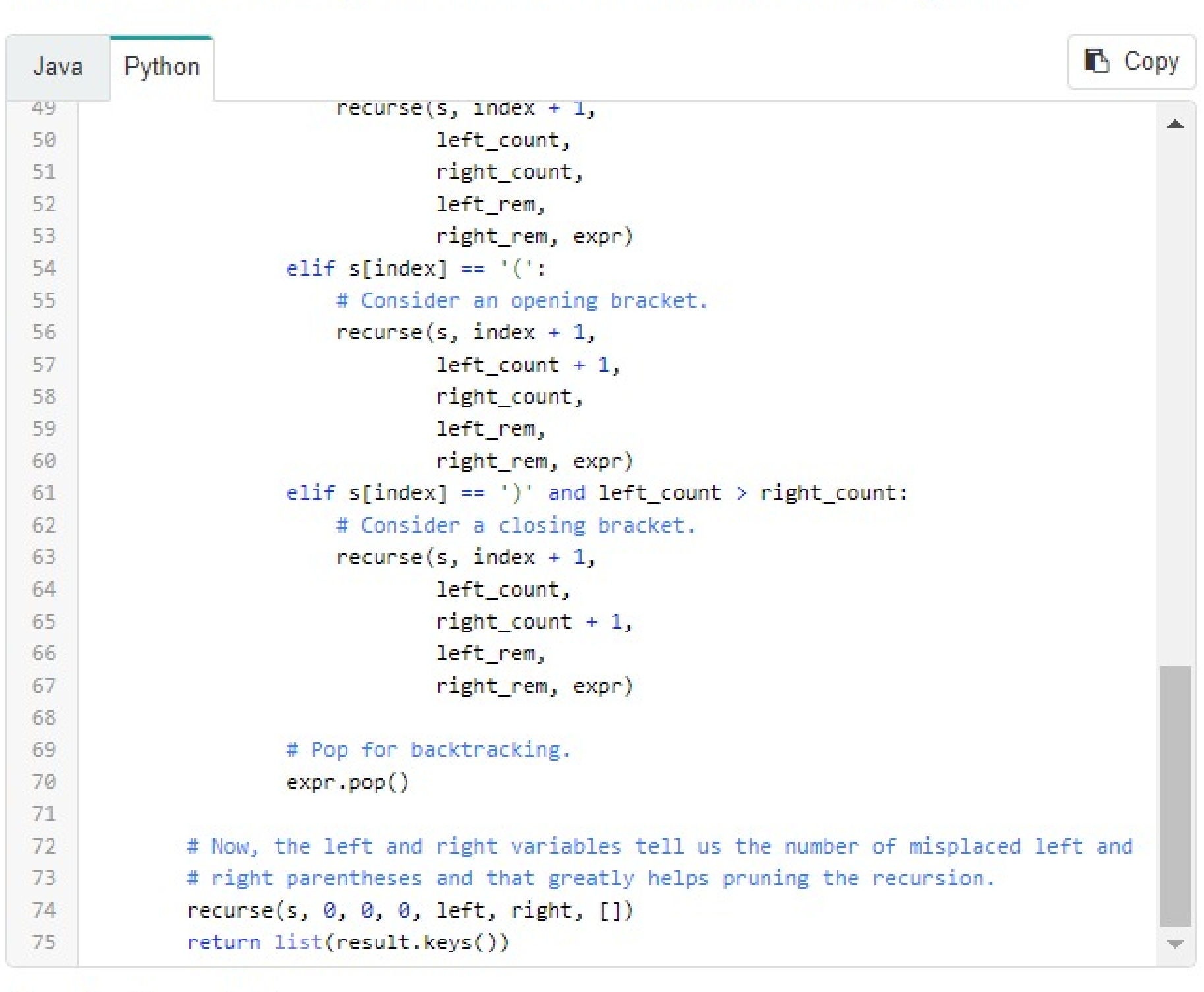
Algorithm

The overall algorithm remains exactly the same as before. The changes that we will incorporate are listed below:

- The state of the recursion is now defined by five different variables:
    1. `index` which represents the current character that we have to process in the original string.
    2. `left_count` which represents the number of left parentheses that have been added to the expression we are building.
    3. `right_count` which represents the number of right parentheses that have been added to the expression we are building.
    4. `left_rem` is the number of left parentheses that remain to be removed.
    5. `right_rem` represents the number of right parentheses that remain to be removed. Overall, for the final expression to be valid, `left_rem == 0` and `right_rem == 0`.
- When we decide to not consider a parenthesis i.e. delete a parenthesis, be it a left or a right parentheses, we have to consider their corresponding remaining counts as well. This means that we can only discard a left parentheses if `left_rem > 0` and similarly for the right one we will check for `right_rem > 0`.
- There are no changes to checks for **considering** a parenthesis. Only the conditions change for **discarding** a parenthesis.
- Condition for an expression being valid in the base case would now become `left_rem == 0 and right_rem == 0`. Note that we don't have to check if `left_count == right_count` anymore because in the case of a valid expression, we would have removed all the misplaced or invalid parenthesis by the time the recursion ends. So, the only check we need if `left_rem == 0 and right_rem == 0`.

> The most important thing here is that we have completely gotten rid of checking if the number of parentheses removed is lesser than the current minimum or not. The reason for this is we always remove the same number of parentheses as defined by `left_rem + right_rem` at the start of recursion.

Now let us look at the implementation for this modified version of algorithm.



```
                    recurse(s, index + 1,
                            left_count,
                            right_count,
                            left_rem,
                            right_rem, expr)
                elif s[index] == '(':
                    # Consider an opening bracket.
                    recurse(s, index + 1,
                            left_count + 1,
                            right_count,
                            left_rem,
                            right_rem, expr)
                elif s[index] == ')' and left_count > right_count:
                    # Consider a closing bracket.
                    recurse(s, index + 1,
                            left_count,
                            right_count + 1,
                            left_rem,
                            right_rem, expr)

                # Pop for backtracking.
                expr.pop()

        # Now, the left and right variables tell us the number of misplaced left and
        # right parentheses and that greatly helps pruning the recursion.
        recurse(s, 0, 0, 0, left, right, [])
        return list(result.keys())
```

Complexity analysis

- Time Complexity : The optimization that we have performed is simply a better form of pruning. Pruning here is something that will vary from one test case to another. In the worst case, we can have something like `((((((((` and the `left_rem = len(S)` and in such a case we can discard all of the characters because all are misplaced. So, in the worst case we **still** have 2 options per parenthesis and that gives us a complexity of $O(2^N)$.
- Space Complexity : The space complexity remains the same i.e. $O(N)$ as previous solution. We have to go to a maximum recursion depth of N before hitting the base case. Note that we are not considering the space required to store the valid expressions. We only count the intermediate space here.

Rate this article: ★ ★ ★ ★ ★

Previous Next

Comments: 42

Sort By

Type comment here... (Markdown is supported)

Preview Post

yewenpu ★ 86 September 24, 2019 12:37 AM

well these kinds of problems are worst when up-front you do not know if a DP i.e. O(poly(n)) solution exist or not. Often times you get stuck for an hour trying to think of a clever sub-problem formulation only to realise the best you can do is exponential in time. What a shame.

86 Share Reply

SHOW 13 REPLIES

sfdye ★ 862 October 4, 2018 4:13 PM

My python implementation, shorter and easier to understand.

Approach 1 (936ms)

```
class Solution:
```

Read More

71 Share Reply

SHOW 6 REPLIES

traceroute ★ 172 March 4, 2019 11:25 PM

To be honest, DFS is not suitable for this kind of problem, to be specific, minimum removals. It's more of concept of layers. Too much depth control logic for dfs. Try BFS.

53 Share Reply

SHOW 2 REPLIES

RameshThaleia ★ 69 April 23, 2020 9:25 PM

There is NO WAY I would be able to figure this out during a Facebook phone interview.

52 Share Reply

wilderfield ★ 90 January 31, 2020 3:01 PM

I hate this problem...

29 Share Reply

park29 ★ 354 September 12, 2019 10:04 PM

what a lengthy solution.

34 Share Reply

lucasmonsteer ★ 32 January 8, 2019 2:01 AM

Could you give an exmaple why

```
            # Pop for backtracking.
            expr.pop()
```

7 Share Reply

SHOW 3 REPLIES

AcidRain ★ 6 April 19, 2020 5:52 AM

It's kinda counter-intuitive that the output for `")("` is `[""]` instead of just `[]`

5 Share Reply

niteshcse14 ★ 4 December 4, 2019 9:55 AM

My C++ implementation, shorter and easier to understand, less than 100.00% of C++ online submissions

```
class Solution {
```

Read More

4 Share Reply

SHOW 2 REPLIES

nits2010 ★ 623 July 20, 2019 12:36 AM

Brillant

3 Share Reply

‹ 1 2 3 4 5 ›