Approach 1: Using a Stack and String Builder

**Intuition**

Let's start by looking at what it means for the parentheses of a string to be *valid*.

The parentheses in a string are balanced *if and only if* these 2 conditions are met:

1. There are the same number of "(" and ")" in the string.
2. Scanning through the string from left to right and counting how many "(" and ")" there are so far, there should never be a time where there are *more* ")" than "(". We call count("(") - count(")") the *balance* of the string..

Here's a simple pseudocode algorithm that checks for these conditions by scanning through the string and incrementing balance each time it sees a "(", and decrementing each time it sees ")". If at any point the balance is negative, which can only happen if we've seen more ")" than "(", then it returns false. If we get to the end, then it returns true only if the balance is 0, which means we've seen the same number of "(" as ")".

function is\_balanced\_parentheses(string)

balance = 0

for each char in the string

if char == "("

balance = balance + 1

if char == ")"

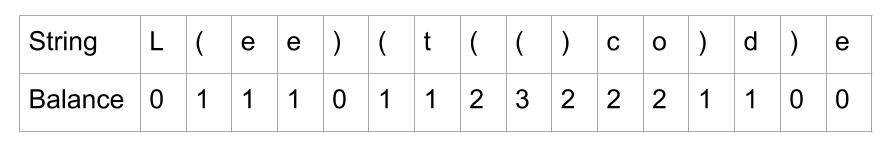
balance = balance - 1

if balance < 0

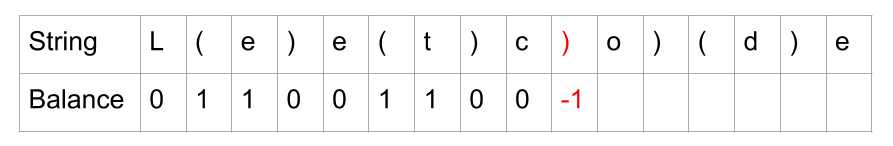
return false

return balance == 0

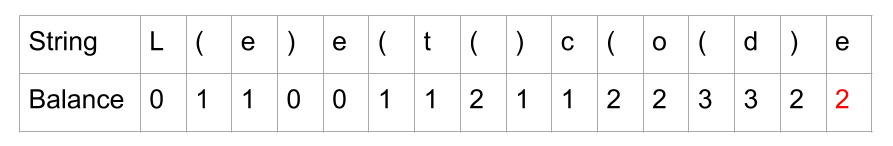
For example, "L(ee)(t(()co)d)e" is a *balanced* string. We'll use a table to show how the balance changes at each point in the string.



The string "L(e)e(t)c)o)(d)e" is invalid because the *balance goes negative*.

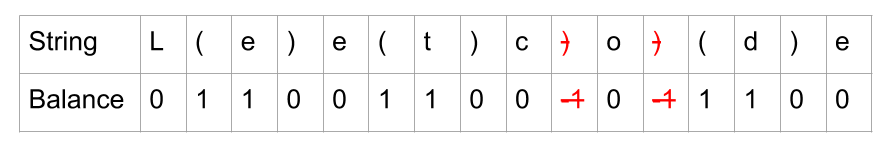


And the string "L(e)e(t()c(o(d)e" is invalid because the balance *is not 0 at the end*.



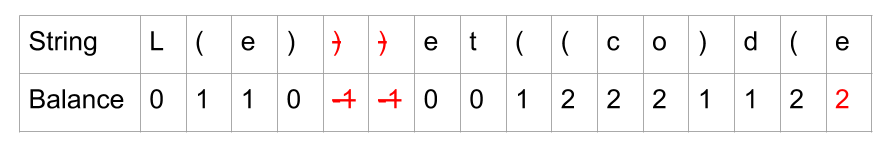
The question asks us to remove the ***minimum number of parentheses*** to make the string valid. So, how can we use the tricks above to achieve this? For starters, we know we'll need to remove any ")" that we encountered when balance was already 0. It would be impossible to remove less ")", because there are not enough "(" before them.

Taking the 2nd example from above, here's what we get when we delete ")" that would have made the balance go negative.



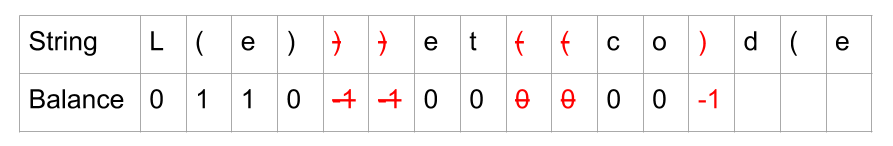
Because we now finish with a zero balance, we know the string is valid.

However, this isn't the full solution. Take a look at this example where we have removed any ")" from "L(e)))et((co)d(e" that would have caused a negative balance, but we still *end with a non-zero balance*.

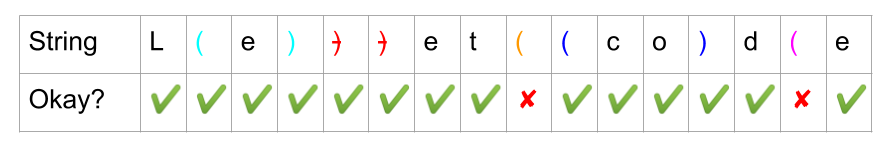


A non-zero balance at the end occurs when there were "(" that were not closed with a ")". Clearly, we'll need to remove some "(" to reduce the balance at the end down to 0. But which should we remove? What will happen if we choose 2 random ones?

Here is the string from above. The 2 "(" we have randomly chosen to remove have been crossed out (along with the 2 ")" from before) and we've checked the balance of the new string.



We've caused the balance to go negative while checking again. Even though we have the same number of "(" and ")" in the string, they don't match up. The last ")" is *before* the last "(". We don't want to do another round of removing ")", because that would no longer be optimal. We need to identify which "(" each of our ")" is actually pairing with. Here is the example with a different color to show each pair. A ")" always pairs with the *closest* "(" that doesn't already have a pair.



The 2 "(" that don't pair with a ")" are the ones we should remove. This way, we know we won't cause a negative balance.

So, remembering that each ")" was paired with the *closest* "(" that isn't already paired, how could we do this in code? We need to know the indexes of the problematic "(".

We can use a **stack**. Each time we see a "(", we should add its index to the stack. Each time we see a ")", we should remove an index from the stack because the ")" will match with whatever "(" was at the top of the stack. The *length of the stack* is equivalent to the balance from above. We will need to do the:

1. Remove a ")" if it is encountered when stack was already empty (prevent negative balance).
2. Remove a "(" if it is left on stack at end (prevent non-zero final balance).

Here is an animation of using a stack to fix the string from above.

1 / 25

You might wonder whether or not this greedy approach is safe, i.e. why not remove an earlier ")" instead? This would "free up" a "(" for the current")" we're looking at, right? The answer is yes, we could do this, but no it won't make any difference to the overall number of ")" we need to remove. This is because whatever "(" was matching with the earlier ")" will now simply match with our current ")" leaving no net benefit.

After removing invalid ")", the number of "(" we remove is the minimum needed to ensure that count("(") == count(")").

**Algorithm**

Let's put all this together into a 2-pass algorithm.

1. Identify all indexes that should be removed.
2. Build a new string with removed indexes.

As explained above, we should use a **stack**. If we put the *indexes* of the "(" on the stack, then we'll know that all the indexes on the stack at the *end* are the indexes of the unmatched "(". We should also use a **set** to keep track of the unmatched ")" we come across. Then, we can remove the character at each of those indexes and then return the edited string.

We need to be really careful with that "removal" step though, as it can be done in O(n)*O*(*n*), but there are many ways of accidentally making it O(n^2)*O*(*n*2). Making these mistakes (and not fixing them) in an interview won't look good. Here's some operations that are O(n)*O*(*n*) that people sometimes assume are O(1)*O*(1).

* Adding or removing (or even changing) just one character *anywhere* in a **string** is O(n)*O*(*n*), because strings are **immutable**. The entire string is rebuilt for every change.
* Adding or removing *not from the end* of a list, vector, or array is O(n)*O*(*n*) because the other items are moved up to make a gap or down to fill in the gap.
* Checking if an item is in a list, because this requires a **linear search**. Even if you use binary search, it'll still be O(\log n)*O*(log*n*), which is not ideal for this problem.

A safe strategy is to iterate over the string and insert each character we want to keep into a **list** (Python) or **StringBuilder** (Java). Then once we have all the characters, it is a single O(n)*O*(*n*) step to convert them into a string.

Recall that checking if an item is in a **set** is O(1)*O*(1). If all the indexes we need to remove are in a set, then we can iterate through each index in the string, check if the current index is in the set, and if it is not, then add the character at that index to the string builder.

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*), where n*n* is the length of the input string.

There are 3 loops we need to analyze. We also need to check carefully for any library functions that are not constant time.

The first loop iterates over the string, and for each character, either does nothing, pushes to a stack or adds to a set. Pushing to a stack and adding to a set are both O(1)*O*(1). Because we are processing each character with an O(1)*O*(1) operation, this overall loop is O(n)*O*(*n*).

The second loop (hidden in library function calls for the Python code) pops each item from the stack and adds it to the set. Again, popping items from a stack is O(1)*O*(1), and there are at most n*n* characters on the stack, and so it too is O(n)*O*(*n*).

The third loop iterates over the string again, and puts characters into a StringBuilder/ list. Checking if an item is in a set and appending to the end of a String Builder or list is O(1)*O*(1). Again, this is O(n)*O*(*n*) overall.

The StringBuilder.toString() method is O(n)*O*(*n*), and so is the "".join(...). So again, this operation is O(n)*O*(*n*).

So this gives us O(4n)*O*(4*n*), and we drop the 44 because it is a constant.

* Space complexity : O(n)*O*(*n*), where n*n* is the length of the input string.

We are using a **stack**, **set**, and **string builder**, each of which could have up to n characters in them, and so require up to O(n)*O*(*n*) space.

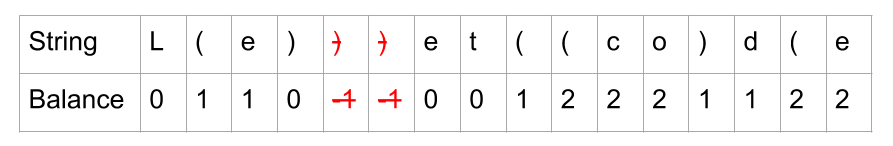
When checking your own implementation, watch out for any O(n)*O*(*n*) library calls *inside loops*, as these would make your solution O(n^2)*O*(*n*2).

Approach 2: Two Pass String Builder

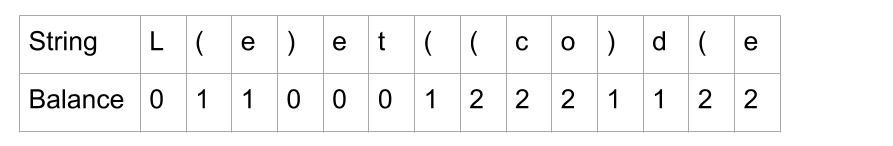
**Intuition**

A key observation you might have made from the previous algorithm is that for all invalid ")", we know immediately that they are invalid (they are the ones we were putting in the set). It is the "(" that we don't know about until the end (as they are what was left on the stack at the end). We could be building up a string builder in that first loop that has *all* of the invalid ")" removed. This would be half the problem solved in the first loop, in O(n)*O*(*n*) time.

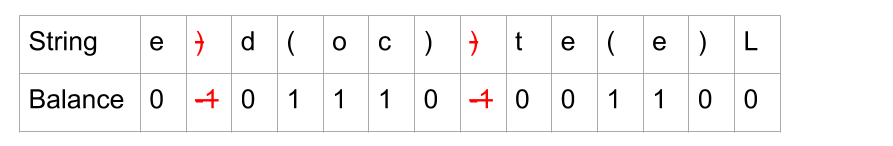
Going back to our example above, we start by identifying all the problematic ")".



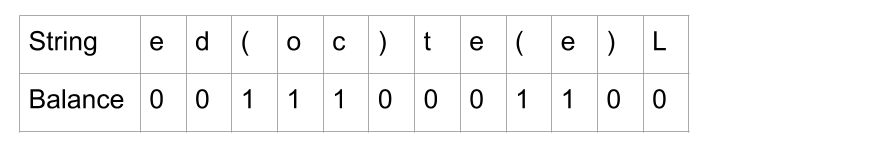
While we were running this pass, we could have been adding all *characters to keep* to a String Builder. This is what we'd have left if we had.

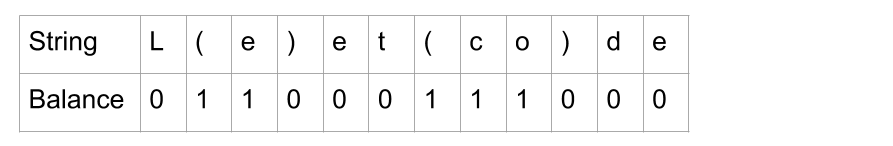


Now, another important observation is that we can use *the same algorithm* to remove the invalid "(". We just need to look at the string in reverse. We do this by swapping the "(" and ")" for each other, and reversing the order of all characters in the string.



So then we can remove those characters, and undo the reverse operation by reversing all characters and swapping "(" and ")" again, and we have the answer!





**Algorithm**

In code, it's best to pull out the common functionality of both passes, otherwise you will have almost the same code repeated twice. A good way to do this is to have a function that takes a string, a symbol to treat as the "open" parenthesis, and a symbol to treat as the "close" parenthesis. The function then returns a string that has all invalid instances of the "closing" symbol removed. Then for the second pass, pass in the reversed string (that was returned from the first pass) and with the "open" and "close" symbols swapped.

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*), where n*n* is the length of the input string.

We need to analyze the removeInvalidClosing function and then the outside code.

removeInvalidClosing processes each character once and optionally modifies balance and adds the character to a string builder. Adding to the end of a string builder is O(1)*O*(1). As there are at most n*n* characters to process, the overall cost is O(n)*O*(*n*).

The other code makes 2 calls to removeInvalidClosing, 2 string reverals, and 1 conversion from string builder to string. These operations are O(n)*O*(*n*), and the 3 is treated as a constant so is dropped. Again, this gives us an overall cost of O(n)*O*(*n*).

Because all parts of the code are O(n)*O*(*n*), the overall time complexity is O(n)*O*(*n*).

* Space complexity : O(n)*O*(*n*), where n*n* is the length of the input string.

The string building still requires O(n)*O*(*n*) space. However, the constants are smaller than the previous approach, as we no longer have the set or stack.

It is impossible to do better, because the input is an immutable string, and the output must be an immutable string. Therefore, manipulating the string cannot be done in-place, and requires O(n)*O*(*n*) space to modify.

When checking your own implementation, watch out for any O(n)*O*(*n*) library functions *inside loops*, as these would make your solution O(n^2)*O*(*n*2).

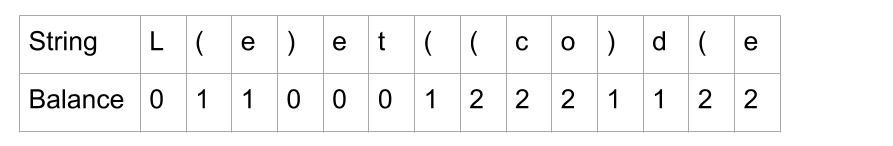
Approach 3: Shortened Two Pass String Builder

**Intuition**

This approach is a simplification of the previous one, and only needs to keep track of the balance. It does not need a stack. Instead of doing the full procedure twice, we can do the first pass and then look at the balance to see how many "(" we need to remove. It turns out that if we remove the *rightmost* '(', we are guaranteed to have a balanced string. So for the second pass, we only need to remove balance "(", starting from the right.

It might be difficult initially to see why this works, so here's a justification.

Consider a string ***s that contains no invalid ")"*** (it has had all the invalid ")" removed by the first pass of the algorithm). It's important to understand that we therefore *know there is a way* of removing balance "(" that *will* make it valid. For example, one of our examples from above.



For a given "(" to be valid, there needs to be *more* ")" *than* "(" *after it* in s (if not, there won't be a ")" leftover for it). If this is true for *all* "(" in s, then s would be valid.

When we remove a "(", all other "(" to the *left* see their ratio of ")" to "(" go up (in other words, they have less others to compete for the ")" with).

So by removing balance "(" from the right, *every* other "(" now has balance less "(" after it, which is the biggest improvement in the ratios we could have possibly got. If any "(" was still not valid after this, then that would mean s had invalid ")" at the start (which it didn't, because it had all of those removed already).

Therefore, this has to be a valid solution.

**Algorithm**

In order to avoid iterating backwards (which adds needless complexity to the algorithm), we also keep track of how many "(" are in the string in the first pass. This way, we can calculate how many "(" we are *keeping*, and count these down as we iterate through the string on the second pass. Then once we've kept enough, we can start dropping them.

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*), where n*n* is the length of the input string.

Same as the above approaches. We have 2 loops that are looping through up to n*n* characters, doing an O(1)*O*(1) operation on each. We also have some O(n)*O*(*n*) library function calls *outside of the loops*.

* Space complexity : O(n)*O*(*n*), where n*n* is the length of the input string.

Like the previous approach, the string building requires O(n)*O*(*n*) space.

When checking your own implementation, watch out for any O(n)*O*(*n*) library functions *inside loops*, as these would make your solution O(n^2)*O*(*n*2).