String problems

String questions involving stacks are popular. String questions that can utilize a stack may involve iterating over the string and putting characters into the stack, and comparing the top of the stack with the current character at each iteration. Stacks are useful for string matching because it saves a "history" of the previous characters.

Example 1: 20 - Valid Parentheses

Given a string `S` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`, determine if the input string is valid. The string is valid if all open brackets are closed by the same type of closing bracket in the correct order, and each closing bracket closes exactly one open bracket.

For example, `s = "({})"` and `s = ""(){}[]` are valid, but `s = "(]"` and `s = "({)}"` are not valid.

The "correct" order is determined by whatever the previous opening bracket was. Whenever there is a closing bracket, it should correspond to the most recent opening bracket. For example, if the string starts `"({["`, and the next 3 characters are closing brackets, then they should be in the order of how recently their opening bracket appeared: `"]})"` (otherwise we would end up with something like `"[)"` occurring). The order is **last in, first out (LIFO)** - the last opening bracket we saw is the first one we should close, which is the perfect functionality for a stack to provide.

As we iterate over the string, if we see an opening bracket, we should put it on the stack. If we see a closing bracket, we can check the most recent unclosed opening bracket by popping it from the top of the stack. If it matches, then continue, if it doesn't or there is no opening bracket on the stack at all (this would occur in a case like `"{}]"`), then we know the string is invalid. In the end, there should be no unmatched open brackets (like in the case of `"(){"`), so the stack should be empty for the string to be valid.

How can we associate the opening and closing brackets together? We can use a hash map to map each opening bracket to its closing bracket. Then when we see a closing bracket, we can use the top of the stack as a key and check if the value is equal to the current character.

```
class Solution:
    def isValid(self, s: str) -> bool:
```

```
        stack = []
        matching = {"(": ")", "[": "]", "{": "}"}

        for c in s:
            if c in matching: # if c is an opening bracket
                stack.append(c)
            else:
                if not stack:
                    return False

                previous_opening = stack.pop()
                if matching[previous_opening] != c:
                    return False

        return not stack
```

Since the stack's push and pop operations are $O(1)$, this gives us a time complexity of $O(n)$, where n is the size of the input array. This is because each element can only be pushed or popped once. The space complexity is also $O(n)$ because the stack's size can grow linearly with the input size.

The key here is recognizing the problem follows a **LIFO** nature.

Example 2: 1047 - Remove All Adjacent Duplicates in String

You are given a string `s`. Continuously remove duplicates (two of the same character beside each other) until you can't anymore. Return the final string after this.

For example, given `s = "abbaca"`, you can first remove the `"bb"` to get `"aaca"`. Next, you can remove the `"aa"` to get `"ca"`. This is the final answer.

The tricky part of this problem is that not all removals are necessarily available at the start. In the example, `"aa"` is only possible **after** deleting the `"bb"`. We don't need to delete the first character until we have already iterated quite a bit past it. This is exacerbated with an example like `s = "abccba"`. The deletion order follows the **LIFO** pattern, where the last (most recent) character is the first one to be deleted.

Therefore we can use a stack. Iterate over the input and put characters in the stack. At each step, if the top of the stack is the same as the current character, we know that they are adjacent (at some point in time) and can be deleted.

```
class Solution:
    def removeDuplicates(self, s: str) -> str:
        stack = []
        for c in s:
            if stack and stack[-1] == c:
                stack.pop()
            else:
                stack.append(c)

        return "".join(stack)
```

The time and space complexity are both $O(n)$, where n is the length of the input. This is because the stack operations are $O(1)$, and the stacks themselves can grow to $O(n)$ size.

Example 3: 844 - Backspace String Compare

Given two strings `s` and `t`, return true if they are equal when both are typed into empty text editors. `'#'` means a backspace character.

For example, given `s = "ab#c"` and `t = "ad#c"`, return true. Because of the backspace, the strings are both equal to `"ac"`.

This problem follows a LIFO pattern, where the first character to be deleted is the one that was most recently typed. We can simulate the typing of the strings with a stack and then compare them at the end.

When typing characters, push them onto a stack. Whatever character is at the top of the stack is the most recently typed character. So when we backspace, we can just pop. Be careful for the edge case where we backspace on an empty string.

```
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        def build(s):
            stack = []
            for c in s:
                if c != "#":
                    stack.append(c)
                elif stack:
                    stack.pop()
```

```
            return "".join(stack)

        return build(s) == build(t)
```

The time and space complexity are linear in the input sizes again.

SEE PROBLEMS 71, 1544